upSuite™ User's Guide

Version 2.6

CC02603-00

9380 Carroll Park Drive
San Diego, CA 92121-2256
858-882-8800
www.ccpu.com

© 2001-2003 Continuous Computing Corporation. All rights reserved.

The information contained in this document is provided “as is” without any express representations of warranties. In addition, Continuous Computing Corporation disclaims all implied representations and warranties, including any warranty of merchantability, fitness for a particular purpose, or non-infringement of third party intellectual property rights.

This document contains proprietary information of Continuous Computing Corporation or under license from third parties. No part of this document may be reproduced in any form or by any means or transferred to any third party without the prior written consent of Continuous Computing Corporation.

Continuous Computing, the Continuous Computing logo, upSuite, upDisk, upBeat, upState, Continuous Control Node (CCN), Continuous System Controller, CCPUnet, CCNtalk, Field Replaceable Microprocessor (FRμ), and Field Replaceable System are trademarks or registered trademarks of the Continuous Computing Corporation or its affiliates. All other product names mentioned herein are trademarks or registered trademarks of their respective owners. The products described in this document maybe protected by U.S. patents, foreign patents, or pending applications. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of Continuous Computing Corporation. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this publication, Continuous Computing Corporation assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

Sun, the Sun logo, SPARCengine, Solaris, and OpenBoot are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

CompactPCI is a registered trademark of PICMG.

The information contained in this document is not designed or intended for use in human life support systems, on-line control of aircraft, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Continuous Computing Corporation disclaims any express or implied warranty of fitness for such uses.

Table of Contents

WELCOME TO UPSUITE HA 1

ABOUT THIS MANUAL 1

Part I: upBeat 3

1 INTRODUCTION TO UPBEAT 5

WHAT IS UPBEAT? 6
WHY SHOULD I USE UPBEAT? 6
HOW DOES UPBEAT WORK? 7
Active and Standby Services *8*
Failover *9*
OPERATION OVERVIEW 10
UBMANAGER 10
SCSIBEAT 11

2 UPBEAT ADMINISTRATION 13

GUIDELINES TO ENSURE HIGH AVAILABILITY 13
Installing software *13*
Inserting or Removing Media *13*
Stopping the system *14*
NETWORK GUIDELINES 14
Routing *14*
MAC Addresses *15*
MONITORING UPBEAT 15
MAINTENANCE 15
TROUBLESHOOTING 16
Links Down *16*
Busy Disks *16*
Operation Problems *16*

3 UPBEAT COMMANDS 17

UPBEAT (ADMIN COMMAND) 17
UPBEAT (STARTUP/SHUTDOWN SCRIPT) 19
UPLICENSE COMMAND 20
UPS COMMAND 21

4 UPBEAT API PROGRAMMER'S GUIDE 23

API OVERVIEW 23
Summary of API Usage *23*
GUIDELINES FOR DESIGNING APPLICATIONS 25
Initialization of upBeat Clients that Provide Services *25*
Registering a Service with upBeat *26*
Polling *28*
Monitoring the state of the system *29*
Using Callback Functions *29*
Error handling *30*
IP Addresses and Failover *31*
Detecting Split Brain Conditions *31*
Strategies for Resolving a Split Brain Condition *32*

5 UPBEAT API REFERENCE 33

CALLBACK FUNCTIONS 34
QUICK REFERENCE OF API FUNCTION CALLS 35
UBACKSVC( ) 36
UBASYNC( ) 37
UBFINI( ) 40
UBGETSTATE( ) 41
UBINIT( ) 43
UBNODE( ) 44
UBNODENAME( ) 45
UBREGSVC( ) 46
UBSERVICEIP( ) 47
UBSETUPPOLLFD( ) 48
UBSVC( ) 49
UBSVCIPPAIR( ) 50
UBSVCNAME( ) 52
UBSVCPEER( ) 53
UBSVCPORT( ) 54
ERROR CODES 55
EINVAL *55*
EIO *55*
EMSGSIZE *55*
ENOTCONN *55*
EPROTO *55*
EPROTONOSUPPORT *55*

6 UPBEAT SAMPLE APPLICATIONS 57

THE wrapper SAMPLE APPLICATION 57
To Use wrapper 58
Source Code of wrapper 58

THE status SAMPLE APPLICATION 64
To Use status 64
Source Code of status 65
THE mt_multi_svc SAMPLE APPLICATION 69
Configuration File *70*
System Architecture *72*
Application Architecture *74*
Termination *75*
To Use mt_multi_svc 76

Part II: upDisk 79

7 INTRODUCTION TO UPDISK 81

WHAT IS upDISK? 81
WHY SHOULD I USE upDISK? 81
HOW DOES upDISK WORK? 82

8 UPDISK ADMINISTRATION 87

upDISK ADMINISTRATOR CONSIDERATIONS 87
Stopping the Solaris machine *87*
Stopping the NFS server *87*
Manipulating underlying file systems *88*
Shutting down the system when a link is up *88*
MONITORING upDISK 89
LOGS 89
TROUBLESHOOTING CONSOLE MESSAGES 89
Bringing Systems Online *89*
Operational Messages *91*
Miscellaneous Transmission Errors *94*
TROUBLESHOOTING: OTHER ISSUES 94
File System Access Denied *94*
File System Out of Disk Space *95*
ipfs Unmount Unsuccessful *95*
Conflicting File Modification Times (mtime) *96*
Repair *96*
Split Brain Conditions *97*
Failover Problems *99*
svc_max_msg_size Causes Problems when Adding CCPUdisk Package *99*

9 FAILOVER 101

upDisk Failover Considerations 101
Normal Failover 101
Double Failures *102*
File Locking *102*
Managed Failover 102
Boot Sequence 104

10 HIGH AVAILABILITY NFS (HA NFS) 107

How Can upDisk Aid My NFS Server? 107
Two HA NFS Architectures *109*
File Handle Persistence During Failover 109
Building an HA NFS Server 110
Edit the configuration file 111
Share the dataset in /etc/ipfstab 112
Configure clients to run upBeat (optional) 113
NFS over UDP vs. TCP 114
Sharing: Subdirectories vs. File Systems 115

11 THE IPFSTAB FILE 117

ipfstab settings 117
ipfs Mount Options 121
Options for Returning to the Application *121*
Maximum Operations Option *123*
Memory Allocation Option *123*
Standby Throttle Option *124*
Synchronous/Asynchronous Option *125*
Standby Modification Time Option *125*

12 UPDISK COMMAND REFERENCE 127

udactive 128
udrepair 131
udstat 132
updisk (admin command) 137
updisk (startup script) 139

13 UPDISK API 141

API Overview 141
Function Call Guidelines 141
Sample API Code 141

14 UPSUITE CONFIGURATION FILE (UPSUITE.CONF) 145

THE CONFIGURATION FILE 145
Managing the Configuration File on Multiple Machines *146*
Using XML in the Configuration File *146*
Avoiding Name Conflicts in the Configuration File *146*
EDITING UPSUITE.CONF 147
<UPSUITECONFIG> TAG 148
<UPBEAT> TAG 148
<NETWORK> TAG 149
NAME attribute *149*
DESCRIPTION attribute *149*
<NODE> TAG 149
NAME attribute *150*
NODE_ID attribute *150*
DESCRIPTION attribute *150*
<INTERFACE> subtag of the <NODE> tag *150*
<PARTITION> subtag of the <NODE> tag *151*
<HEARTBEAT> TAG 152
NAME attribute *152*
TYPE attribute *152*
TIMEOUT_MSEC attribute *153*
RESEND_MSEC attribute *153*
<NODE_REF> subtag of <HEARTBEAT> *153*
<LINK> subtag of <HEARTBEAT> *153*
<SERVICE> TAG 154
NAME attribute *155*
SERVICE_ID attribute *155*
TYPE attribute *155*
STARTUPDELAY_SEC attribute *155*
PORT attribute *155*
<NODE_REF> subtag of the <SERVICE> tag *155*
<LINK> subtag of the <SERVICE> tag *156*
<SCSIBEAT> subtag of the <SERVICE> tag *157*
<HANFS> subtag of the <SERVICE> tag *157*
<SERVICE_IP> subtag of the <SERVICE> tag *158*
CONFIGURATION FILE EXAMPLE 159
SAMPLE CONFIGURATIONS 161
Configuration A *161*
Configuration B *166*
Configuration C *170*
Partition Table *175*
Engineering Guidelines for Upsuite *176*

Part III: ubManager 177

15 INTRODUCTION TO UBMANAGER 179

WHAT IS UBMANAGER? 179
HOW DOES UBMANAGER WORK? 180
Service Groups *180*
Resource Monitoring *182*
Application Process Launching and Monitoring *183*
Failover Scripting *183*
ubManager Components *185*

16 UBMANAGER COMMAND REFERENCE 187

SHELL COMMANDS 187

17 UBMANAGER MONITORS 203

MONITORS OVERVIEW 203
LOADMON 205
RPCMON 206
UBPINGER 207

18 UBMANAGER SEMICOLON-DELIMITED CONFIGURATION FILE 209

COMMENTS 209
VERSION 209
GROUP DEFINITION 210
MONITOR DEFINITION 210
APPLICATION DEFINITION 212
SAMPLE SEMICOLON-DELIMITED CONFIGURATION FILE 212

TYPOGRAPHIC CONVENTIONS 215

GLOSSARY 217

TECHNICAL SUPPORT 219

INDEX 221

Welcome to upSuite HA

upSuite HA™ is a high-availability software product for Solaris 7, 8, 9 and 10 that instantly provides high availability for any application or server. upSuite HA detects software, hardware, or network failure and provides sub-second failover to a standby system. Real-time data replication at the filesystem level enables upSuite to provide rapid failover and recovery. upSuite can provide these services transparently to the application and operating system. No APIs are necessary.

- Sub-second failure detection and failover
- Runs on TCP/IP LAN
- Tunable heartbeats control failover
- Efficient application-transparent failover
- Efficient repair and replay recovery
- Minimal use of CPU and bandwidth
- Installs in minutes
- Simple to implement and maintain active/standby architecture

About This Manual

This manual describes the following software units that make up upSuite:

- Part I: upBeat
- Part II: upDisk
- Part III: ubManager

In this manual it is assumed that you are familiar with:

- The Solaris operating system
- TCP/IP and networking
- XML notation
- C programming

Part I: upBeat

upBeat is a failure detection manager that monitors the activity and state of components over an IP network by sending “heartbeats” between components. upBeat can detect node, interface/link, or application failures. If a failure is detected, upBeat initiates failover in less than one second, making upBeat well suited to environments in which high availability is required.

upBeat can be used alone or as part of upSuite HA™ and is available for Solaris 7, 8, 9 and 10 on SPARC.

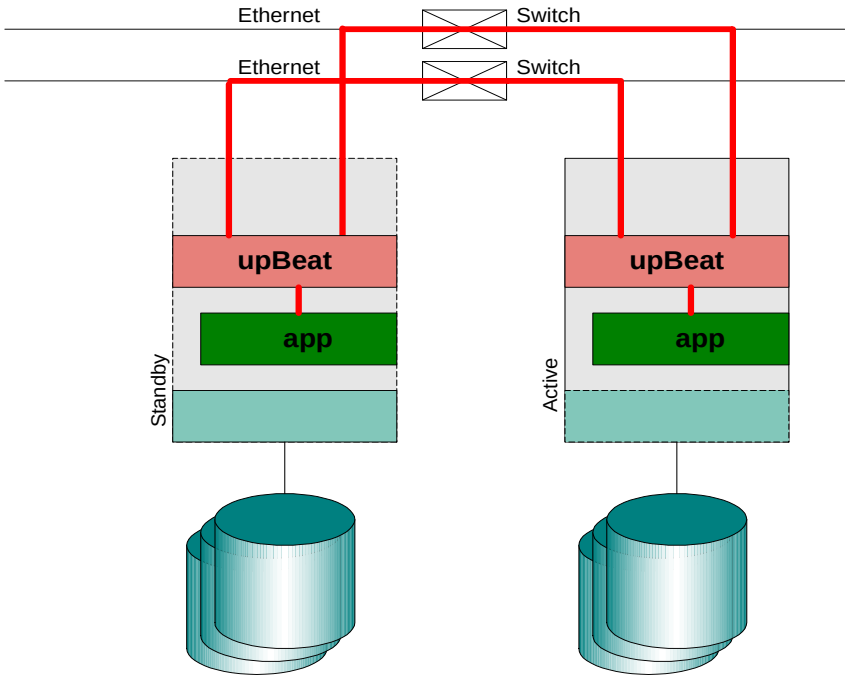

Figure 1 upBeat network operation

1 Introduction to upBeat

This chapter introduces you to upBeat, an architecture for managing heartbeats among clients, servers, and services as part of upSuite HA™, a series of software modules designed to provide transparent high-availability capability for applications.

In this chapter

What is upBeat? 6
Why Should I Use upBeat? 6
How Does upBeat Work? 7
Failover 9
ubManager 10
SCSIbeat 11

What is upBeat?

upBeat is a failure detection manager that monitors the activity and state of components over an IP network. upBeat detects node, interface/link, or application failures and initiates failover.

The three major functions of upBeat are:

1. Manages and initiates failover
2. Shares information with upBeat running on other components
3. Shares data with interested local applications.

upBeat is designed for failure detection and failover in an IP network environment.

Other features of upBeat include:

- Heartbeats
- Server/server monitoring allows a standby server to take over in case of failure
- Server/service monitoring notifies a server as to which services need to be removed to a standby server
- Client/server monitoring notifies clients which servers are up and which services are active on each server
- Immediate disk failure notification

upBeat is available for Solaris 7, 8, 9 and 10 on SPARC.

Why Should I Use upBeat?

upSuite HA provides rapid and flexible failure detection over an IP network. By providing sub-second detection and response initiation, upSuite HA can decrease failover time to less than a second.

Specific benefits of upBeat include:

- Creation of a heartbeat framework is unnecessary
- Heartbeats can be added over additional media
- Applications can register with the heartbeat manager
- Failover scenarios support flexible configurations

How Does upBeat Work?

upBeat sends “heartbeats,” similar to ping packets, over the networks between components, or on private networks between processors, to verify that they are active and healthy. Heartbeats are used to detect node, interface/link, or application failures, in which case upBeat initiates a failover.

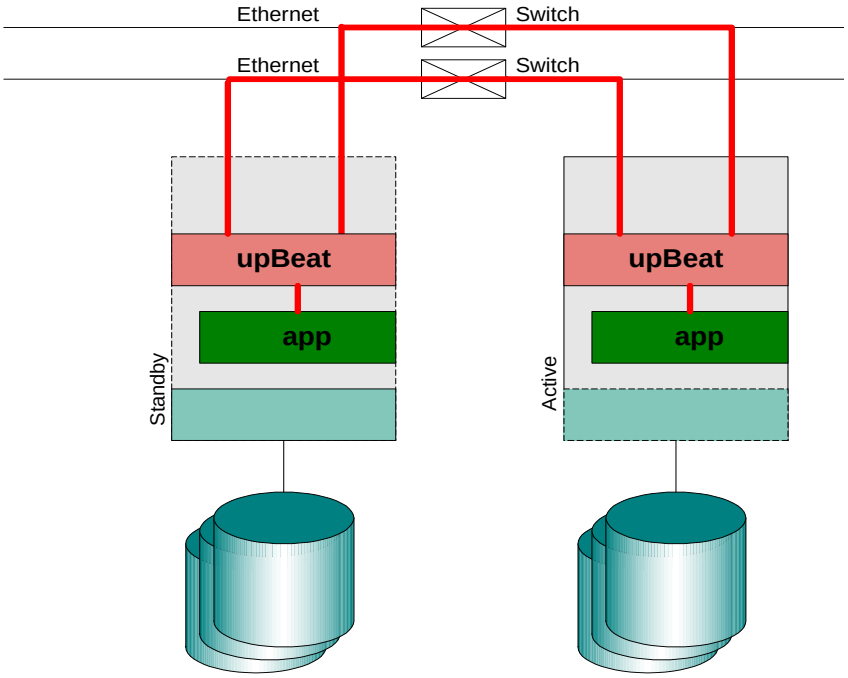

Figure 1 upBeat network operation

upBeat uses two types of notification elements which help to govern failover and communication between components:

1. Node-to-node heartbeats are sent back and forth between instances of upBeat running on different processors to verify the status of the processors and to share information between them.

2. Client-registration and notification requests about the state of the system are made by a resident application through upBeat.

Active and Standby Services

An instance of upBeat runs on each node in the system. Each upBeat instance interacts with every other upBeat configured in the configuration file (`upsuite.conf`; for more information, see “upSuite Configuration File (upsuite.conf)” on page 145). There may be any number of nodes running upBeat and any number of upBeat clients monitoring the network.

However, any particular service may reside on at most two nodes: on one node it will be in the standby state, while on the other node it can be either in the standby or active state. On any node where multiple services are running, there may be some services that are active and some that are standby. Thus a service, not a node, is referred to as being either “active” or “standby.”

When an application registers to provide a service, the application can inform upBeat whether it would prefer to start off as the active or the standby. upBeat may honor that request or not depending on the state of the rest of the system. Once a service application is active, it can register with upBeat to become standby if it is no longer able to provide its service. Alternatively, a service application can refuse an active directive from upBeat if it is unable to provide the service. (Refer to “Guidelines for Designing Applications” on page 25 for more detailed information about initialization and registration).

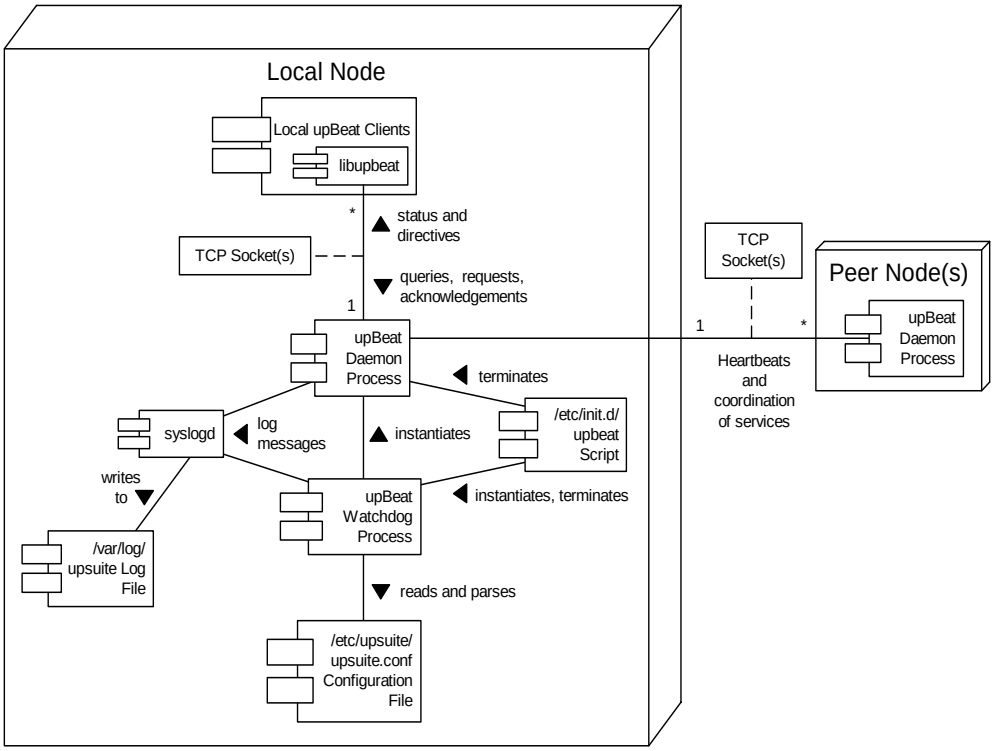

Figure 2 upBeat operations sequence

Failover

upBeat manages failover between two services. If the active service unexpectedly terminates, or it re-registers requesting a standby role, upBeat sends the standby service a directive to become active.

If necessary, upBeat migrates the service IP address to the new active server (this is known as “IP failover”) by:

- Deconfiguring the service IP on the old active (if it is alive)
- Configuring the service IP on the new active
- Sending a gratuitous ARP so that systems and routers on the local subnet immediately communicate with the new active

Operation Overview

Upon startup, upBeat spends a brief period of time discovering the state and health of the network. This period is configurable; the default is five seconds. During this time, upBeat establishes heartbeats, listens to other upBeat servers, and listens to local applications; this information is then shared with other upBeat servers. However, upBeat does not tell local applications anything about the network, and upBeat does not assign any service roles during this time.

After this period, upBeat notifies local applications about which services are up or down on which nodes (servers), and starts assigning roles.

Finally, upBeat processes its services according to the following logic:

1. If there is already an active service, the local service is assigned standby (even if it is requesting active).

2. If the service is requesting active, and there is not already an active service, it will be assigned active.

3. If a service requests to be active on two nodes, upBeat instructs one to be active and the other to be standby.

4. If no nodes are requesting active, then one of the nodes requesting standby may be assigned active.

Note: If it is inappropriate for a node to become active, it is the responsibility of the service, not upBeat, to decline.

ubManager

ubManager (pronounced “You-Be-Manager”) is an additional package available for use with upBeat (but is not part of the upBeat installation). ubManager is designed to enhance the failure detection abilities of upBeat. To that end, ubManager is ideal for situations in which monitoring upBeat clients is required.

With ubManager you can monitor a single service or a set of services that you have defined as a group. In doing so, you can monitor the health of the system and, according to the conditions you have configured, determine whether or not a given system or service may remain the active one.

ubManager relies on service monitors to learn about the health of the system, essentially using upBeat as a database to determine whether a system can remain active. (Monitor processes may be scripts or binary executables). Each service monitor is designed to watch a particular

element of the system and report status back to ubManager via upBeat.

ubManager can support five types of service monitors: permanent, transient, periodic, “smart” or “dumb.” Permanent, transient, and periodic monitors function exactly as their names imply. Smart monitors are able to directly update service status in upBeat, while dumb monitors rely on ubManager to perform this function, based on the exit status of the given service.

SCSIbeat

SCSIbeat, a component of upBeat, monitors disks and reports failures. Solaris may spend a long time attempting retries on a failed disk before reporting the event to disk-dependent applications; SCSIbeat, however, notifies upBeat of disk failures promptly. upBeat then initiates disk failover, designating disk-dependent applications as standby.

When a disks fails or becomes unresponsive, SCSIbeat tells upBeat. If there is an active service that depends on that disk, upBeat tells the service to go standby, i.e., a failover is initiated.

SCSIbeat monitors disk statistics to make sure the disk is operating. SCSIbeat also sends periodic commands to the disk; this ensures there will be some activity on an otherwise idle disk, and it allows SCSIbeat to discover disk errors. These commands involve little overhead and provide rapid feedback about the disk’s health. You can adjust how often the command is sent, and the timeout before SCSIbeat informs upBeat that the disk is not responding (for more information, see “<NODE> tag” on page 149).

SCSIbeat monitors disk partitions, because Solaris provides programmatic access to disks by providing programmatic access to partitions. To allow for different disk partition names on each node, you can specify a name for each disk partition by using the the <PARTITION> subtag of the <NODE> tag in the upSuite configuration file. You can then use the <SCSIBEAT> tag to configure a given service to depend on the partition by name. For more information about these tags, see “upSuite Configuration File (upsuite.conf)” on page 145.

If you use Solaris DiskSuite to mirror disks which you want to monitor with SCSIbeat, monitor the DiskSuite device (for example, `/dev/md/rdsk/d3`), not the underlying disk device (for example, `/dev/rdsk/c0t0d0s3`). You must also specify a long timeout period to avoid unnecessary failover. Solaris DiskSuite takes approximately three seconds to detect a disk failure and detach the disk. If the SCSIbeat timeout (set in the TIMEOUT_MSEC attribute of the <NODE> tag) is less than this, SCSIbeat will fail and force a failover unnecessarily. In order to avoid this and failover only if both disks fail, set TIMEOUT_MSEC to 3500 msec or more, depending on the specific performance of your disk drive.

2 upBeat Administration

This chapter details issues specific to system administrators.

Guidelines to Ensure High Availability

Certain system administration activities can prevent even the highest-priority real-time processes in the system from running. If you are administering a high-availability system with aggressive timeouts, we recommend you avoid these types of interaction with a running high-availability machine whenever possible. The following activities should be undertaken with care or avoided:

- Installing software
- Inserting or Removing Media
- Stopping the system

The rest of this section explains each of these areas of potential concern in more detail.

Installing software

The **`drvconfig`**, **`add_drv`**, or **`pkgadd`** commands temporarily lock up the system and cause upBeat heartbeats to fail. This results in a split-brain condition. (Note that **`drvconfig`** can be run manually or it may be run as a result of **`add_drv`** or **`pkgadd`**, in which the package installed contains a kernel module, such as a device driver). Therefore, do not run **`drvconfig`**, **`add_drv`**, or **`pkgadd`** on your system. If you need to install or upgrade software:

1. Shut down upSuite on the standby system.
2. Install the software on the standby system.
3. Restart upSuite on the standby system.
4. Perform a failover.
5. Repeat steps 1-3 on the formerly active system.

Inserting or Removing Media

Inserting or removing media, especially from SCSI devices, can lock up the system and cause upBeat heartbeats to fail. This results in a split-brain condition. The Solaris SCSI disk driver

can block the processor (busy wait) for a long time in certain cases when CDs, CDRWs, or MOs are inserted or ejected.

In a conservative high availability system, it is therefore inadvisable to use removeable media.

If, however, your system availability is not as critical, using very long timeouts can provide some protection against going into a split-brain condition when removing media.

Stopping the system

Stopping the system (for example, using break, L1-A, or Setup-A) and then continuing will cause heartbeats to fail, resulting in a split-brain condition.

The two most typical reasons for stopping a system are to use the kernel debugger (which is not normally necessary in a healthy production system) and to prepare to reboot a hung system. Stopping the system should not cause a problem, but stopping and then continuing will do so.

Network Guidelines

In this chapter we detail issues specific to those upBeat users designing complex network architectures.

Routing

By default, upBeat only communicates with other nodes on the same subnet by using the SO_DONTROUTE socket option. This means you are limited to systems that are connected by hubs and switches. This behavior guarantees network paths thereby allowing upBeat to reliably detect network failures.

You can change this default behavior and establish a heartbeat between nodes that are on different subnets or that are separated by routers. Packets will be routed according to your node's routing tables. You must ensure there are separate network paths for each link and that packets for one link cannot be routed over a path for another link. Incorrect routing tables may create failure detection problems or a single point of failure, leaving your system vulnerable to a split brain condition.

If you choose to route, you must ensure that your routing tables are set up so that packets intended for one network do not get routed to a different network. If this happens, there could be undetected (latent) failures which could eventually lead to system downtime due to network outage without warning. Therefore, ensure that you have independent paths to each network and that routers do not route packets between the two networks.

For more information about how to enable routing, see "<HEARTBEAT> tag" on page 152.

MAC Addresses

Solaris, by default, instructs all Ethernet interfaces in the same system to use the same MAC address. If each of the interfaces is connected to a different network, this works well. However, if you connect several LANs to the same hub or switch (or cascade of hubs and switches), network slowdowns or outages may result. In particular, if the hubs and switches are the kind that "learn" on which port a MAC address is located, more problems may occur.

If you segment a switch into virtual LANs (VLANs), be aware that some switches incorrectly route among VLANs based on the MAC address. As mentioned before, these switches have the ability to "learn" (albeit incorrectly in some cases) MAC addresses.

Listed below are possible solutions to this problem:

- Set up Solaris so that it does not instruct all Ethernet hardware to use the same address by modifying the `local-mac-address?` and OpenBoot variable.
- Use only switches that do not have "leaky" VLANs.
- Avoid the use of VLANs.

Continuous Computing Corporation recommends setting the `local-mac-address?` variable to "true" on all Sun systems running upSuite HA.

Monitoring upBeat

Current status and changes to upBeat's links, services, or nodes can be monitored via the `/opt/upsuite/bin/ups` program. You can learn whether links, services, or nodes are `UP` or `DOWN` (the only possible states listed with the output message). See "upBeat Commands" on page 17 for further explanation and usage of this command.

Running upBeat in debug mode (**`/usr/sbin/upbeat -d`**) allows you to monitor upBeat operations. See "upbeat (admin command)" on page 17.

As an alternative, you may also use the log file, `/var/log/upsuite`, to monitor historical operations of upBeat.

Maintenance

upBeat's status can be monitored via the `syslog`. By default, the upBeat daemon sends all output to the `syslog` facilty, unless upBeat is running in debug mode, in which case it sends all output to the terminal.

Troubleshooting

This section describes some common issues that arise when using upBeat and tells how to respond to these events.

Links Down

If `ups` reports a link is down, try pinging that link manually. If there is no answer to the ping, check your cabling and switches.

If you find the link fluctuating between `UP` and `DOWN`, check your heartbeat `TIMEOUT_MSEC` setting in `upsuite.conf`. Ensure that this setting is not too short for your network.

Busy Disks

If you encounter the message below, your disk is not responding:

```
Oct  3 09:52:41 left upbeat[418]: disk or partition '/dev/
   rdsk/c0t0d0s0' is not responding
```

If investigation reveals no problems with your disk or SCSI bus, you may have your SCSIbeat settings too low in `upsuite.conf`. Increase the values of the TIMEOUT_MSEC and FREQ_MSEC attributes beneath the <PARTITION> subtag of the <NODE> tag. See “<NODE> tag” on page 149 for more detailed information about these settings.

Operation Problems

If you find error messages in `/var/log/upsuite`, ensure that your system configuration as you have defined it in the configuration file (`upsuite.conf`) matches your actual configuration. You can do this by using the **`ifconfig -a`** command at the Solaris prompt. Match the **`ifconfig -a`** output information against that in your configuration file.

3 upBeat Commands

This chapter describes upBeat's commands and their uses. The commands are organized alphabetically and include usage, a description, options, relevant files, and other related commands. You must be logged in as superuser (**root**) to use these commands.

In this chapter

upbeat (admin command) 17
upbeat (startup/shutdown script) 19
uplicense command 20
ups command 21

upbeat (admin command)

NAME

upbeat

USAGE

/usr/sbin/upbeat [[-c] [-d] [-s | -S] [-v]] [-h] [-V]

DESCRIPTION

This command should not normally be run directly; it is run from the **upbeat** startup/shutdown script as part of system startup. If you want to start or stop upBeat manually, use the **upbeat** startup/shutdown script.

This command starts the watchdog and upBeat daemons. If you were to run `ps -ef | grep upbeat`, you would see two upBeats running; these are the upBeat processes. One process acts as watchdog, and will restart the main upBeat process (the daemon) if it stops unexpectedly.

OPTIONS

Running this command without any arguments causes upBeat to run as a daemon. Program messages are sent to `syslog`.

Including the **`-c`** flag causes program messages to be sent to your console in addition to any other place they are being sent, for example, to `syslog`.

Including the **`-d`** flag runs upBeat in debug mode, so that upBeat runs in the foreground. Messages that would normally go to `syslog` will output to the terminal along with additional debugging information.

The **`-s`** flag forces program messages to be sent to `syslog` even if, for example, you have already run the command with the **`-d`** flag.

The **`-S`** flag causes program messages to not be sent to `syslog` even if you have already run a command which would normally route them there.

The **`-v`**(erbose) flag increases the amount of detail in your output messages, regardless of where they are being sent.

The **`-h`**(elp) flag prints the usage of this command.

The **`-V`**(ersion) flag prints the version of upBeat.

FILES

`upsuite.conf`

SEE ALSO

“upbeat” on page 19

upbeat (startup/shutdown script)

NAME	**`upbeat`**
USAGE	**`/etc/init.d/upbeat {start | stop}`**
DESCRIPTION	This command is run automatically upon Solaris reboot and invokes **`/usr/sbin/upbeat`**.
OPTIONS	Including the **`start`** argument allows you to begin using upBeat after you have installed it without rebooting. The **`stop`** argument gracefully shuts down `upbeat`. (When Solaris gracefully shuts down it uses this command and argument.) Use this argument if you are having a problem and want to debug. Then run **`/usr/sbin/upbeat -d`**. When you are done debugging, run **`/etc/init.d/upbeat start`**.
FILES	`upsuite.conf`
SEE ALSO	“upbeat” on page 17

uplicense command

NAME	**uplicense**
USAGE	**/usr/sbin/uplicense [-v | -o] [-f *filename*]**
DESCRIPTION	Run this command after you have installed your licenses to validate your upSuite HA software. This command is run at boot time with the **-o** flag by /etc/init.d/uplicense.
OPTIONS	Including the **-o** (overwrite) flag translates the license file (usually /etc/upsuite/license) into a binary file (.dat) which the upSuite HA programs use and a text file (.txt) which allows you to verify the number of valid licenses you have. Running this command overwrites previously generated .dat and .txt license files (but not /etc/upsuite/license). This is typically what you would run when upgrading from older versions of upSuite HA. Including the **-v** (verify) flag causes a check to be run on the system to ensure the licenses are present in /etc/upsuite/license. If no arguments are specified, this is the default command. The **-f *filename*** argument enables you to run the program in a test environment without affecting the system. For example, you could run **/usr/sbin/uplicense -f mytestfile** for a system other than the one on which you are running the command.
FILES	/etc/upsuite/license
SEE ALSO	Refer to the *Installation Guide* for license installation instructions.

ups command

NAME	**ups**
USAGE	**/opt/upsuite/bin/ups [-q]**
DESCRIPTION	This command gives status information for upBeat. The output from this command notifies you of all available links, nodes, and services of which upBeat is aware.
OPTIONS	Including the **-q** flag will run this command once, print out status, and finish. Otherwise, `ups` runs continuously, alerting you to all changes in links, nodes, and services.
FILES	`upsuite.conf`
SEE ALSO	“upbeat” on page 17 “upbeat” on page 19

4 upBeat API Programmer's Guide

This chapter contains an introduction to upBeat's Application Programming Interface (API), and gives information about how to use the API calls together in an application.

For details on the syntax and use of each API call, see "upBeat API Reference" on page 33. For complete example programs, see "upBeat Sample Applications" on page 57.

In this chapter

API Overview 23
Guidelines for Designing Applications 25

API Overview

By using the upBeat API, you can request upBeat to send status information about the network to your application. In addition, the application can be notified whenever the state of the network changes. Notification lets the application know when any change occurs in the status of links, nodes, and services.

You can also register services with upBeat through the API. Individual services may request active or standby status, and upBeat grants or denies the requested status according to its knowledge of the network. As changes occur on the network, upBeat dynamically designates components as active or standby to maintain high availability of the application.

The API can be organized into the following categories:

- Initialization and termination activities
- Normal operation, including service registration and polling for application events
- Error handling

The API library is defined in the files `libupbeat.a` and `libupbeat.h`.

Summary of API Usage

The minimum that an application must do to use the upBeat API, and thus become an upBeat client, is as follows: the very first upBeat API call it makes must be to `ubInit()`, which returns a handle used for the rest of the API calls. When the application terminates, or when it is finished using upBeat, the very last call must be to `ubFini()` to terminate the application's registration with upBeat for the local node and to free the resources allocated for the application by `ubInit()`.

The following guidelines provide a summary of the suggested minimum use of the upBeat API in a client application. For a more detailed discussion, see "Guidelines for Designing Applications" on page 25.

- All programs must call `ubInit()` before they call any other function in the upBeat API.
- Programs should call `ubAsync()` periodically to gather system information. The best way to do this is is to use `ubSetupPollfd()` together with the UNIX API call `poll()` or `select()`. A single-threaded application can add the upBeat `fd` to its main `poll()` or `select()` call; a multi-threaded application can dedicate a thread to `ubAsync()`.
- To register a service, call `ubRegSvc()`.
- If the upBeat client application is sufficiently complex, it may be beneficial to allocate a separate thread specifically for interacting with upBeat.
- Multi-threaded application callbacks *must* lock data. Otherwise, functions may be performed in undesired sequences, thus damaging your data.
- All programs must call `ubFini()` before they exit.
- All applications must include the following header file:

  ```
  #include <libupbeat.h>
  ```

- Applications should be compiled and linked as follows:

  ```
  gcc -I/opt/upsuite/include -c -o myapp.o myapp.c
  gcc -L/opt/upsuite/lib -o myapp myapp.o -lupbeat
  ```

  Multithreaded applications:

  ```
  gcc -D_REENTRANT -I/opt/upsuite/include -c -o myapp.o myapp.c
  gcc -L/opt/upsuite/lib -o myapp myapp.o -lupbeat -lthread
  ```

  or

  ```
  gcc -L/opt/upsuite/lib -o myapp myapp.o -lupbeat -lpthread
  ```

`libupbeat` is not gcc-specific. All programs incorporating `libupbeat`'s function calls will compile on any Sun SPARC C compiler.

Guidelines for Designing Applications

This section contains a detailed discussion of how to use the upBeat API calls in an application. It covers the following topics:

- Initialization of upBeat Clients that Provide Services
- Registering a Service with upBeat
- Polling
- Monitoring the state of the system
- Using Callback Functions
- Error handling
- IP Addresses and Failover
- Detecting Split Brain Conditions
- Strategies for Resolving a Split Brain Condition

Initialization of upBeat Clients that Provide Services

This section describes how upBeat clients that provide services use the upBeat API during initialization.

The first call the application makes to the upBeat API is to `ubInit()`. This begins the initialization. However, `ubInit()` will block until the upBeat startup delay period set in the configuration file has expired. For more information about setting the startup delay, see "<UPBEAT> tag" on page 148.

During initialization, the client needs to know its own service name or service ID (as specified in `upsuite.conf`). This information can be hard-coded into the program, passed as a command-line argument, or provided using whatever technique you prefer. If a service knows only the name of the service it provides, it can call `ubSvc()` to find its corresponding service ID.

To find out its local node ID, an upBeat client can call `ubNode()`. The client can also use `ubNode()` to find out the node ID of any peer node, if it knows the name of the peer node. Alternatively, the client can call `ubSvcPeer()` with its own service ID to find the node ID where its peer service is running.

If the service has any <LINK> tags in the configuration file, the application can call `ubSvcIPPair()` iteratively to get pairs of IP addresses for the local and peer nodes. Typically, these pairs of addresses are used for the service applications on the local and peer nodes to communicate with each other.

If the service has a TCP or UDP port number configured in `upsuite.conf`, the application can call `ubSvcPort()` to get the port number its service uses in its socket binding. If the service has a floating, or migrating, IP address configured in `upsuite.conf`, the application can call `ubServiceIP()`to get the address.

The last upBeat API call made during an upBeat client's initialization activities will most likely be `ubGetState()`. This call provides state information to the client. Up to three callback functions are called as a result of the call to `ubGetState()`. A pointer to a data structure is passed to `ubGetState()` that gives the functions, if any, to call back. If any of the following three callback functions are defined, they will be called:

- The node callback function is called once for each peer node, giving the peer node's state.
- The link callback function is called once for each link, giving the link's state.
- The service callback function is called once for each service, giving the node on which the service is active, if any.

For more information, see "Callback Functions" on page 34.

Registering a Service with upBeat

If an upBeat client is to provide a service, it must call `ubRegSvc()` to register the service with upBeat. A service may request active or standby status upon registering, and upBeat grants or refuses this request according to its knowledge of the network. As changes occur on the network, upBeat dynamically designates a service as active or standby accordingly.

Typically, upBeat registers as active the first service that requests active status. Any peer service that later attempts to register as active is designated as standby. The service will know whether it is currently active or standby once its directive callback function is called. upBeat client applications can monitor the states of other services by using their service callback functions.

For more information, see "Callback Functions" on page 34.

Directives from upBeat

upBeat will send a directive to a service (using the directive callback function) in two situations:

- **A service registers for active or standby status.**
  A service registering for standby status will get a directive to go standby. A service registering for active might get a directive to go either active or standby, depending on the system conditions.

- **The peer service is active and registers for standby or acknowledges an active directive with standby status.**
  The service will get a directive to go active.

If a service gets a directive to go standby, it must assume that it is in standby mode at that point. Similarly, if a service gets a directive to go active, and it can go active, it can assume that it is active at that point. If a service gets a directive to go active, but is unable to do so for any reason, it must acknowledge the directive with a standby status, using `ubAckSvc()`, and can assume that it remains standby.

Typical service registration as active or standby

In a typical situation, two peer services registering as active upon instantiation. As indicated previously, the first service to register as active will get a directive to go active and its peer will get a directive to go standby. Both services acknowledge the directive by calling `ubAckSvc()`. Normally, a service directed to go active will acknowledge the directive with an active status, and a service directed to go standby will acknowledge the directive with a standby status.

Alternatively, peer services can initially register as standby, and then later re-register as active depending on events. If both of the peer services are in standby mode, they will remain so until one of them registers as active. In this situation, upBeat will not send a directive to go active unless a service requests active status by calling `ubRegSvc()`.

If a service is active and then registers for standby status by calling `ubRegSvc()`, upBeat will send it a directive to go standby and send its peer a directive to go active.

Handling of various service registration situations

The previous section describes how service registration typically occurs. However, it is possible for clients to issue registration or acknowledgement calls that are different from those that would typically be expected.

For example, suppose a service is given a directive to go standby, but instead of acknowledging and accepting the directive, the service acknowledges the directive with an active status. In this case, the acknowledgement is ignored by upBeat, and the service must assume that it is standby.

If a standby service, without being given a directive, calls `ubAckSvc()`, that unexpected acknowledgement is ignored. Likewise, if an active service calls `ubAckSvc()` with active status, without being given a directive, the acknowledgement is ignored.

However, if an active service calls `ubAckSvc()` with standby status, without being given a directive to go standby, the service will automatically be given standby status. Its peer, if running, will get a directive to go active.

If a service is given a directive to go active, and the service acknowledges the directive with a standby status, its peer service will get a directive to go active. If the peer service also acknowledges its active directive with standby status, the first service will again get a directive to go active. This "ping-ponging" effect will continue for a total of three times before upBeat stops giving the services an active directive. That is, if a service acknowledges an active directive with standby status three times in a row, upBeat will stop giving the service an active directive and the service remains in standby. If both of the peer services are still in standby mode at this point, one of them must register for active status before an active directive will again be sent to it.

Polling

After an upBeat client finishes its initialization activities and perhaps registers a service, it typically enters an event handling loop. The UNIX API call `poll()` or `select()` is used to block until data is sent to the application to be processed.

To receive information from upBeat asynchronously, the client should call `ubSetupPollfd()` each time before calling `poll()` or `select()`. Calling `ubSetupPollfd()` will properly set up a `pollfd` data structure in case the client's connection to the upBeat daemon is lost and then reestablished. The `pollfd` data structure is used directly by `poll()`, and its file descriptor can be extracted for use by `select()`.

After `ubSetupPollfd()` is called and then `poll()` is called and returns, the client should call `ubAsync()`, regardless of the returned events for the associated `pollfd` data structure. Similar to `ubGetState()`, `ubAsync()` calls up to four callback functions. A pointer to a data structure is passed to `ubAsync()` that gives the functions, if any, to call back.

The callback functions can include any of the following:

- The node callback function is called whenever a change to a peer node's state occurs. Typically, when all of the links to a peer node are down, this callback function is called to indicate that the peer node is down. When a link comes back up, this function is called to indicate that the node is back up.
- The link callback function is called whenever a change to the state of a link to a peer node changes—the link has either gone down or has come up.
- The service callback function is called whenever a change to the state of a service occurs—either the service has become active or has become standby.
- For an upBeat client that provides a service, the directive callback function is called whenever upBeat wants to initialize or change the state of the service—either the service is to go active or to go standby. The service must acknowledge a directive with `ubAckSvc()`.

For more information, see “Callback Functions” on page 34.

Monitoring the state of the system

The upBeat API provides two functions that you can use to find out the state of the system: `ubAsync()` and `ubGetState()`.

- `ubAsync()` gets asynchronous status information from upBeat. An application must call `ubAsync()` periodically to keep `libupbeat` up to date; using `poll()` is the recommended way.
- `ubGetState()` gets synchronous status information stored in `libupbeat`. An application may call `ubGetState()` anytime, but this is generally unnecessary; the state information will be as recent as the most recent call to `ubInit()` or `ubAsync()`.

Using Callback Functions

Callback functions are functions defined within your application. They run when when the upBeat daemon has notified your application of a change of status and your application calls `ubGetState()` or `ubAsync()`. The callback functions come in four types. The node, link, and service callback functions are used to provide status information about nodes, links, and services. The directive callback function is used to tell a service to assume a given status, active or standby.

Your application can define some or all of these types of callback functions. An upBeat client that acts only as a network monitor defines one or more of the `node`, `link`, and `service` types of callback functions. An upBeat client that provides a service must have the `directive` type of callback function and may, optionally, define any of the others.

The node, link, and service callback functions are passed an argument that indicates whether the callback function is being called as a result of a call to `ubAsync()` or as a result of a call to `ubGetState()`. In this way, the callback functions can perform slightly different behavior depending on which function called them. For more information about this and other arguments to the callback functions, see “ubAsync( )” on page 37.

For an overview of the different types of callback functions, see “Callback Functions” on page 34.

Using the link callback function

The `link` callback function indicates whether a connection is available to the peer by returning UP or DOWN. To keep a mapping between the local and peer nodes’ interfaces, use `ubSvcIPPair()`.

The link callback function’s `ub_link_id_t` argument is the IP address of a peer node’s

interface and may be cast to an `in_addr_t`.

The following example shows an implementation of a link callback function:

```
static void
link_callback(void *argp, ub_link_id_t link, ub_status_t status, boolean_t async)
{
       struct in_addr in;
       in.s_addr = (in_addr_t)link;

       printf("link_callback: link = %s, status = %s, called by = %s.\n",
                inet_ntoa(in),
                UB_STATUS_UP == status ? "UP" : "DOWN",
                async ? "ubAsync()" : "ubGetState()");
}
```

When calling `inet_ntoa()`, be aware that it returns a pointer to a buffer that will change the next time `inet_ntoa()` is called. Therefore, if you need to make multiple calls to `inet_ntoa`, copy the contents of this buffer to your program's local buffer storage before each subsequent call.

Using the service callback function

When called by `ubGetState()`, the service callback function's `ub_node_id_t` argument will be zero if the specified service is not yet active on any node. The service callback function's `ub_status_t` argument indicates the specified service's active/standby state. If the argument is UB_STATUS_DOWN, the specified service either does not exist or is in standby on the specified node. If the argument is UB_STATUS_UP, the specified service is active on the specified node. Note that a service callback function will be called indicating the state of all services, not just for the service that an upBeat client may have registered for.

Error handling

If an upBeat API call fails, the application should call `ubFini()` and cease high-availability operations, at least until a successful call to `ubInit()` is made.

Many of the upBeat API functions indicate an error by returning `-1` or NULL. When this occurs, the value of `errno` is also set. This value can be set in two ways:

- The upBeat API call
- An underlying function or system call that was made by the upBeat API

If an underlying function or system call failed, the value set by that call is preserved. Otherwise, the upBeat API call sets `errno` itself. There is no way to detect whether an upBeat API or an underlying call set the `errno` value. You can use the `strerror()` function to get an

error string for an error.

It is not possible to list all the values that might be set by underlying calls. The error values that can be set by the upBeat API are detailed in “Error codes” on page 55.

IP Addresses and Failover

When a service is active, any floating (or migrating) IP address that is configured for the service in `upsuite.conf` is brought up on the node for which the service is active. The corresponding floating IP address is brought down on the node on which the service becomes standby.

When a failover occurs, any TCP connections to the floating IP address are lost, and clients receive errors when they try to send data on those connections. In this situation, clients should close and reestablish their connections.

Detecting Split Brain Conditions

If two servers lose heartbeat connectivity but both are still running, there is a network partition, referred to throughout upSuite documentation as a split-brain condition. If the same service is running on both of the servers, both services will become active; each service can no longer detect the heartbeat from the other server, and so must assume that the other service has failed.

Once heartbeat connectivity is restored, it is possible to detect and correct the dual-active condition. If an application is active (in response to the directive callback function) and later hears (through the service callback function) that its peer is active, then a dual-active condition has been detected.

The rest of this section examines the sequence of events in detail.

Consider the following scenario: a system has two nodes; all of the links to one node go down; later, a link comes back up.

1. For each upBeat client on each node, the link callback function is called for each link that has gone down.

2. After the final link callback is called, the node callback function is called, indicating that the peer node is down.

3. upBeat sends the standby services a directive to go active.

4. If a service is active and its standby peer service goes active when directed, a split-brain condition occurs. Each side loses communication with the other, and each service can accept clients on its portion of the now isolated network.

If a link comes up, the following occurs:

1. Each upBeat client's link callback function is called, indicating the link has come up.

2. If the service is active and is implemented in such a way that it will always accept the active role when directed to, the service can assume a split-brain condition.

3. After the link callback functions are called, the node callback functions are called, indicating that the peer node has come back up.

4. All of the service callback functions are called, indicating which services on the peer node are active.

5. If a service is active and its service callback function indicates that its peer service is also active, the service has positively detected a split-brain condition.

Strategies for Resolving a Split Brain Condition

How a service resolves a split brain condition depends on the application. You can devise any strategy best suited to your configuration needs. For example:

- Issue a message indicating that operator intervention is required.
- Always have the service running on the node with the lower (or higher) node ID value remain active while the other goes standby.
- Have the service that most (or least) recently became active go standby.
- Design your code to prefer the system that is most reliable, most accessible, or has the most memory.

5 upBeat API Reference

This chapter contains an alphabetical listing of the function calls in the upBeat API.

In this chapter

Callback Functions 34
Quick Reference of API Function Calls 35
ubAckSvc( ) 36
ubAsync( ) 37
ubFini( ) 40
ubGetState( ) 41
ubInit( ) 43
ubNode( ) 44
ubNodeName( ) 45
ubRegSvc( ) 46
ubServiceIP( ) 47
ubSetupPollfd( ) 48
ubSvc( ) 49
ubSvcIPPair( ) 50
ubSvcName( ) 52
ubSvcPeer( ) 53
ubSvcPort( ) 54
Error codes 55

Callback Functions

The following table lists the major upBeat daemon callback functions used by the upBeat API. For additional information, see “ubAsync( )” on page 37.

Callback	Description
node	Gives the state of a node. This function is called as a result of a call to `ubAsync()` or `ubGetState()`. `ubAsync()` calls this function whenever a change of state occurs to a node with which the current node shares a heartbeat. When `ubGetState()` is called, this function is called for every node in the configuration except the current node. The node the software is running on is always UP since the software is running.
link	Gives the state of a link. This function is called as a result of a call to `ubAsync()` or `ubGetState()`. `ubAsync()` calls this function whenever a change of state occurs to an interface of any node with which the current node shares a heartbeat. When `ubGetState()` is called, this function is called for every link in the configuration.
service	Gives the state of a service including the node, if any, on which the service is active. This function is called as a result of a call to `ubAsync()` or `ubGetState()`. `ubAsync()` calls this function whenever a change of state occurs to a service. When `ubGetState()` is called, this function is called for every service in the configuration. If one of the arguments is node ID 0, the status will be down, meaning that there is no active server.
directive	Instructs a service to assume a given status, either active or standby. This function is called as a result of a call to `ubAsync()`. `ubAsync()` calls this function whenever upBeat wants to initialize or change the active/standby state of a service registered by this program on this node.

Table 1 Callback functions

Quick Reference of API Function Calls

Initialization and Termination	
ubInit( )	Initialize a connection to the local upBeat daemon.
ubFini( )	Shut down a connection to upBeat and free any resources allocated by `ubInit()`.
ubAckSvc( )	Respond to a UB_ACTIVE or UB_STANDBY directive.
ubAsync( )	Receive asynchronous status information from upBeat.
ubGetState( )	Get synchronous state stored in `libupbeat`.
ubRegSvc( )	Register to provide a service.
ubSetupPollfd( )	Set up a poll file descriptor structure so that the application can `poll()` on `libupbeat`'s connection to the local upBeat daemon.
ubNode( )	Get the node ID that corresponds to a given node name.
ubNodeName( )	Get the name of a node.
ubServiceIP( )	Get one of the IP addresses associated with a given service.
ubSvc( )	Get the service ID of a named service.
ubSvcIPPair( )	Get a pair of IP addresses on the same network for a pair of servers configured to provide a service.
ubSvcName( )	Get the name of a service.
ubSvcPeer( )	Get the node ID of the other server that is offering a service.
ubSvcPort( )	Get the TCP or UDP port number for the service.

Table 2 Quick reference of function calls

ubAckSvc( )

NAME
ubAckSvc

SYNOPSIS

Respond to a UB_ACTIVE or UB_STANDBY directive.

```
int ubAckSvc(upbeat_t *upbeatp, ub_svc_id_t service,
  ub_svc_status_t svc_status)
```

upbeatp A handle as returned by `ubInit()`.

service A numeric service ID corresponding to the SERVICE_ID attribute for a <SERVICE> tag in the upSuite configuration file.

svc_status UB_ACTIVE or UB_STANDBY.

DESCRIPTION

The application uses `ubAckSvc()` in its directive callback to respond to a UB_ACTIVE or UB_STANDBY directive from upBeat. The application may agree with upBeat, or it may downgrade the directive from UB_ACTIVE to UB_STANDBY, but it may not upgrade the directive from UB_STANDBY to UB_ACTIVE.

RETURN VALUES

`ubAckSvc()` returns 0 if it successfully sends the message to upBeat; otherwise it returns -1 to indicate failure. In most high-availability applications, failure should be treated as a catastrophic failure, and the application should shut down.

ERRORS

See “Error codes” on page 55.

SEE ALSO

“ubRegSvc( )” on page 46

ubAsync( )

NAME
ubAsync

SYNOPSIS

Receive asynchronous status information from upBeat.

```
int ubAsync(upbeat_t *upbeatp, const ub_ops_t *opsp)
```

upbeatp A handle as returned by `ubInit()`.

opsp Pointer to a `ub_ops_t` structure

DESCRIPTION

The application is required to call `ubAsync()` periodically. `ubAsync()` can run in its own thread, or it can be called from an application thread. If `libupbeat`'s connection to the local upBeat daemon is ever lost (which is very unlikely), `ubAsync()` will attempt to reestablish the connection.

The **opsp** argument points to a `ub_ops_t` structure which gives upBeat the client's node, link, service, and directive callback functions (if any). The structure also contains the argument passed to the callback functions. The `ub_ops_t` structure is defined as follows:

```
typedef struct ub_ops {
  void   *arg;
  void   (*node)(void *arg, ub_node_id_t node_id,
         ub_status_t node_status, boolean_t async);
  void   (*link)(void *arg, ub_link_id_t link_id,
         ub_status_t link_status, boolean_t async);
  void   (*service)(void *arg, ub_svc_id_t service_id,
         ub_node_id_t node_id, ub_status_t service_status,
         boolean_t async);
  void   (*directive)(void *arg, ub_svc_id_t service_id,
         ub_svc_status_t svc_status);
} ub_ops_t;
```

The arguments are as follows:

arg Any pointer. Can be NULL. Passed as the first argument to each of the callback functions.

node_id Corresponds to the NODE_ID attribute of the <NODE> tag in the upSuite configuration file (`upsuite.conf`).

link_id An IP address. Can be cast to an **`in_addr_t`**.

service_id Corresponds to the SERVICE_ID attribute of the <SERVICE> tag in the upSuite configuration file (`upsuite.conf`).

node_status One of the following values: UB_STATUS_UP or UB_STATUS_DOWN. If the callback function was called by `ubAsync()`, UB_STATUS_UP indicates that a peer node's upBeat has started sharing heartbeats with the local node, and UB_STATUS_DOWN indicates that such sharing has stopped. If the callback function was called by `ubGetState()`, UB_STATUS_UP indicates that a peer node's upBeat is sharing heartbeats with the local node, and UB_STATUS_DOWN indicates that no such sharing is underway.

link_status One of the following values: UB_STATUS_UP or UB_STATUS_DOWN. If the callback function was called by `ubAsync()`, UB_STATUS_UP indicates that a link between a peer node and the local node has started sharing heartbeats, and UB_STATUS_DOWN indicates that such sharing has stopped. If the callback function was called by `ubGetState()`, UB_STATUS_UP indicates that a link between a peer node and the local node is sharing heartbeats, and UB_STATUS_DOWN indicates that no such sharing is underway.

service_status

One of the following values: UB_STATUS_UP or UB_STATUS_DOWN. If the callback function was called by `ubAsync()`, UB_STATUS_UP indicates that an upBeat service has become active on a node, and UB_STATUS_DOWN indicates that an upBeat service has become standby on a node. If the callback function was called by `ubGetState()`, UB_STATUS_UP indicates that an upBeat service is active on a node, and UB_STATUS_DOWN indicates either that an upBeat service is standby on a node, or does not exist on that node.

If **`service_status`** is UB_STATUS_DOWN and **`node_id`** is 0, the

service is not active on any node.

svc_status Specifies what role a service should take as directed by upBeat. One of the following values: UB_STANDBY or UB_ACTIVE. UB_STANDBY indicates that the service must become standby. UB_ACTIVE indicates that the service may become active; the service can also refuse and retain its standby status by using ubAckSvc() with its svc_status argument set to UB_STANDBY, as described in "ubAckSvc( )" on page 36.

async B_TRUE if the callback function was called by ubGetState(); the state is synchronous. B_FALSE if the callback function was called by ubAsync(); the state is asynchronous. The use of this argument allows the same callbacks to be used by both ubAsync() and ubGetState().

node() A pointer to the node callback function. For more information, see "Callback Functions" on page 34.

link() A pointer to the link callback function. For more information, see "Callback Functions" on page 34.

service() A pointer to the service callback function. For more information, see "Callback Functions" on page 34.

directive() A pointer to the directive callback function. For more information, see "Callback Functions" on page 34.

RETURN VALUES

ubAsync() returns -1 if the connection to the local upBeat daemon is lost and cannot be reestablished. Returns 0 otherwise. In most high-availability applications, failure should be treated as a catastrophic failure, and the application should shut down.

ERRORS

See "Error codes" on page 55.

SEE ALSO

"ubGetState" on page 41
"ubSetupPollfd" on page 48
"Callback Functions" on page 34

ubFini( )

NAME
ubFini

SYNOPSIS

Shut down a connection to upBeat and free any resources allocated by `ubInit()`.

`void ubFini(upbeat_t*)`

`upbeatp` A handle as returned by `ubInit()`.

DESCRIPTION

`ubInit()` allocates resources and establishes a connection to the local upBeat daemon.
`ubFini()` shuts down the connection and frees the resources.

RETURN VALUES

None.

ERRORS

None.

SEE ALSO

“ubInit” on page 43.

ubGetState( )

NAME
`ubGetState`

SYNOPSIS

Get synchronous state stored in `libupbeat`.

`int ubGetState(upbeat_t *upbeatp, const ub_ops_t *opsp)`

`upbeatp` A handle as returned by `ubInit()`.

`opsp` Pointer to an `ub_ops_t` structure.

DESCRIPTION

The **`opsp`** argument points to a `ub_ops_t` structure which gives upBeat the client's node, link, service, and directive callback functions (if any). For more information about this structure, see "ubAsync" on page 37.

An application may call `ubGetState()` to get a snapshot of the state from `libupbeat`. The information included in this snapshot is whatever state was stored after the last call to `ubInit()` or `ubAsync()`, and is therefore not necessarily up to date with the current state of the system. The snapshot includes the state of the nodes defined in the upSuite configuration file (`upsuite.conf`) with which the current node shares a heartbeat, the links associated with those nodes, and the services associated with those nodes.

The node, link, and service callbacks will be called once for each item. This is different from the behavior of `ubAsync()`, which only calls the callbacks if state has changed.

Typically, an application should call `ubGetState()` only during startup, just after calling `ubInit()`. Thereafter, the application must use `ubSetupPollfd()` and `ubAsync()` to keep informed.

RETURN VALUES

`ubGetState()` returns `-1` if the connection to the local upBeat daemon is lost and cannot be reestablished; `0` otherwise. In most high-availability applications, failure should be treated as a catastrophic failure, and the application should shut down.

ERRORS

See "Error codes" on page 55.

SEE ALSO

"ubAsync" on page 37

"Callback Functions" on page 34

ubInit( )

NAME
`ubInit`

SYNOPSIS

Initialize a connection to the local upBeat daemon.

```
upbeat_t* ubInit()
```

DESCRIPTION

The application must call `ubInit()` before calling other upBeat API functions. `ubInit()` allocates resources and establishes a connection to the local upBeat daemon. Calls to `ubInit()` will block until the upBeat startup delay period set in the configuration file has expired.

When finished, the application must call `ubFini()` to shut down the connection and free resources allocated by `ubInit()`.

RETURN VALUES

On success, `ubInit()` returns an upBeat handle that is to be passed to all other API functions; on failure, `ubInit` returns `errno`.

ERRORS

`ubInit()` fails if it cannot allocate resources or if it cannot establish a connection to the local upBeat daemon, and sets `errno`. See “Error codes” on page 55.

SEE ALSO

“ubFini( )” on page 40

ubNode( )

NAME
ubNode

SYNOPSIS

Get the node ID that corresponds to a given node name.

```
ub_node_id_t ubNode(upbeat_t *upbeatp, char *nodename)
```

upbeatp A handle as returned by `ubInit()`.

nodename The NAME attribute of a <NODE> tag in the upSuite configuration file (`upsuite.conf`).

DESCRIPTION

Get the `ID` from the upSuite configuration file for a node.

If the value of **nodename** is `NULL`, `ubNode()` returns the `ID` of the current node.

RETURN VALUES

`ubNode()` returns the node `ID` (a positive integer) if the handle passed in **upbeatp** is not NULL and it finds the node; otherwise it returns `0`.

ERRORS

None.

SEE ALSO

“ubNodeName( )” on page 45

ubNodeName( )

NAME
ubNodeName

SYNOPSIS

Get the name of a node.

```
char *ubNodeName(upbeat_t *upbeatp, ub_node_id_t node)
```

upbeatp A handle as returned by `ubInit()`.

node Node ID of the node for which you want to get the name. Use the NODE_ID attribute of a <NODE> tag in `upsuite.conf`, or 0 for the current node.

DESCRIPTION

Get the name of a node from the upSuite configuration file. Also useful for translating a node ID in a node, link, or service callback.

RETURN VALUES

If the handle passed in **upbeatp** is not NULL, the node exists, and the node's NAME attribute has a value, `ubNodeName()` returns the name; otherwise, returns NULL.

ERRORS

None.

SEE ALSO

"ubNode( )" on page 44.

ubRegSvc()

NAME
ubRegSvc

SYNOPSIS

Register to provide a service.

```
int ubRegSvc(upbeat_t*, ub_svc_id_t service,
 ub_svc_status_t svc_status)
```

upbeatp A handle as returned by ubInit().

service A numeric service ID corresponding to the SERVICE_ID attribute for a <SERVICE> tag in the upSuite configuration file (upsuite.conf).

svc_status UB_ACTIVE or UB_STANDBY. Indicates whether the application prefers the service to have active or standby status.

DESCRIPTION

An application uses ubRegSvc() to register to provide a service. The application provides **svc_status** to indicate whether it prefers to be active or standby, but there is no guarantee upBeat will honor the preference. The application can call ubRegSvc() again later for the same service to try to change the active/standby status of a service for which it has already registered.

Later, upBeat will indicate via the application's directive callback whether the application should assume an active or a standby role. At that time, the application should use ubAckSvc() to confirm or deny the role.

RETURN VALUES

ubRegSvc() returns 0 if it successfully sends the request to upBeat, -1 on failure. Success does not mean that service registration is complete or that the **svc_status** preference has been granted.

ERRORS

See "Error codes" on page 55.

SEE ALSO

"ubAckSvc( )" on page 36

ubServiceIP( )

NAME
`ubServiceIP`

SYNOPSIS

Get one of the IP addresses associated with a given service.

```
int ubServiceIP(upbeat_t *upbeatp, ub_svc_id_t service,
in_addr_t in_addr, ub_service_ip_t *service_ip)
```

`upbeatp`	A handle as returned by `ubInit()`.
`service`	Service ID of the service for which you want to get an IP address. Use an ID from a service callback or one returned by `ubSvc()`.
`in_addr`	IP address returned by the most recent previous call to `ubServiceIP()`, or `0`.
`service_ip`	Out parameter in which a set of data including the IP address found by `ubServiceIP()` is returned.

DESCRIPTION

In the upSuite configuration file, the <SERVICE_IP> subtags of the <SERVICE> tag specify which IP addresses upBeat manages for the service. These are the IP addresses involved in IP failover, and are the addresses at which clients expect to contact the service. You can use `ubServiceIP()` iteratively to get these IP addresses from the configuration file.

RETURN VALUES

If `ubServiceIP()` executes successfully (the handle passed in **`upbeatp`** is not NULL, <SERVICE_IP> subtags are found, and you are not at the end of the list), returns 1; otherwise, returns `0`.

ERRORS

None.

SEE ALSO

“ubSvc( )” on page 49
“ubSvcIPPair( )” on page 50

ubSetupPollfd( )

NAME

ubSetupPollfd

SYNOPSIS

Set up a poll file descriptor structure so that the application can `poll()` on `libupbeat`'s connection to the local upBeat daemon.

```
void ubSetupPollfd(upbeat_t *upbeatp,
  struct pollfd *pollfdp)
```

upbeatp A handle as returned by `ubInit()`.

pollfdp Pointer to a `pollfd` structure.

DESCRIPTION

An application must call `ubAsync()` periodically. One way to do that is to `poll()` on `libupbeat`'s file descriptor, and call `ubAsync()` whenever there are events. `ubSetupPollfd()` fills in a `pollfd` structure with the file descriptor and the required `pollfdp->` events.

An application that uses `select()` can extract the file descriptor from **`pollfdp`**.

An application should call `ubSetupPollfd()` before each call to `poll()` in case `ubAsync()` has reestablished a connection to upBeat.

RETURN VALUES

None.

ERRORS

None.

SEE ALSO

"ubAsync( )" on page 37

ubSvc( )

NAME
ubSvc

SYNOPSIS

Get the service ID of a named service.

```
ub_svc_id_t ubSvc(upbeat_t *upbeatp, char *servicename)
```

upbeatp A handle as returned by **ubInit()**.

servicename The NAME attribute of a <SERVICE> tag in the upSuite configuration file (`upsuite.conf`).

DESCRIPTION

Get the ID from the upSuite configuration file for a service.

RETURN VALUES

`ubSvc()` returns the service ID (a positive integer) if the handle passed in **upbeatp** is not NULL and it finds the service; otherwise it returns `0`.

ERRORS

None.

SEE ALSO

“ubServiceIP( )” on page 47
“ubSvcName( )” on page 52

ubSvcIPPair( )

NAME
ubSvcIPPair

SYNOPSIS

Get a pair of IP addresses on the same network for a pair of servers configured to provide a service.

```
int ubSvcIPPair(upbeat_t *upbeatp, ub_node_id_t node,
 ub_svc_id_t service, in_addr_t in_addr,
 ub_ippair_t *ippairp)
```

upbeatp	A handle as returned by `ubInit()`.
node	Node ID, or 0 for the current node.
service	Service ID.
in_addr	Previous IP address or `0`.
ippairp	Pointer to location where `ubSvcIPPair()` stores the IP addresses.

DESCRIPTION

Services in the upSuite configuration file are configured for certain nodes and networks. This function fetches a pair of IP addresses on the same network, one from each node, and places them in the `ub_ippair_t` pointed to by **ippairp**:

```
typedef struct ub_ippair {
   in_addr_t   local;
   in_addr_t   remote;
ub_ippair_t;
```

The address for the node specified by the **node** argument is placed in `ippairp->local`; the other address is placed in `ippairp->remote`.

Subsequent calls return additional IP address pairs for the same service on other networks. These pairs of addresses are typically used between two servers providing a service.

The first time `ubSvcIPPair()` is called, **in_addr** should be `0`; on subsequent calls, **in_addr** should be the `ippairp->local` from the previous call. This way an application can retrieve all the pairs of IP addresses for a service. Calling `ubSvcIPPair()` iteratively

returns the IP addresses from the <NODE> tags as indexed by NETWORK.

RETURN VALUES

`ubSvcIPPair()` returns `0` if the handle passed in **`upbeatp`** is NULL, there are no IP pairs, or you are at the end of the list; otherwise, it returns non-zero.

ERRORS

None.

SEE ALSO

“ubServiceIP( )” on page 47

ubSvcName( )

NAME
ubSvcName

SYNOPSIS

Get the name of a service.

```
char *ubSvcName(upbeat_t *upbeatp, ub_svc_id_t service)
```

upbeatp A handle as returned by `ubInit()`.

service Service ID of the service for which you want to get the name. Use an ID from a service callback or one returned by `ubSvc()`.

DESCRIPTION

Get the name of a service from the upSuite configuration file.

RETURN VALUES

If the handle passed in **upbeatp** is not NULL, the service exists, and its NAME attribute has a value, `ubSvcName()` returns the name; otherwise, returns `NULL`.

ERRORS

None.

SEE ALSO

“ubSvc( )” on page 49.

ubSvcPeer( )

NAME
ubSvcPeer

SYNOPSIS

Get the node ID of the other server that is offering a service.

```
ub_node_id_t ubSvcPeer(upbeat_t *upbeatp, ub_node_id_t node,
  ub_svc_id_t service)
```

upbeatp	A handle as returned by `ubInit()`.
node	Node ID of one server offering the service specified by the **service** argument. For the current node, use 0.
service	Service ID.

DESCRIPTION

Get the node ID of the other server that is offering a service. Typically, the **node** argument is the current node as returned by `ubNode(upbeatp, NULL)`.

RETURN VALUES

`ubSvcPeer()` returns the node ID of the other server if the handle passed in **upbeatp** is not NULL, it finds the service, and node is one of the servers; otherwise, it returns `0`.

ERRORS

None.

SEE ALSO

“ubNode( )” on page 44
“ubSvc( )” on page 49

ubSvcPort( )

NAME

`ubSvcPort`

SYNOPSIS

Get the TCP or UDP port number for the service.

```
in_port_t ubSvcPort(upbeat_t *upbeatp, ub_svc_id_t service)
```

`upbeatp` A handle as returned by `ubInit()`.

`service` Service ID.

DESCRIPTION

Get the TCP or UDP port number for the service from the upSuite configuration file.

RETURN VALUES

If the handle passed in **`upbeatp`** is not NULL, the service exists, and it has a PORT attribute, `ubSvcPort()` returns the port number; otherwise, returns `0`.

ERRORS

None.

SEE ALSO

“ubSvcIPPair( )” on page 50

Error codes

Many of the upBeat API functions indicate an error by returning `-1` or `NULL`. When this occurs, the value of `errno` is also set. This value can be set in two ways:

- The upBeat API call
- An underlying function or system call that was made by the upBeat API

It is not possible to list all the values that might be set by underlying calls. This section describes the error values that can be set by the upBeat API.

EINVAL

The handle (an `upbeat_t*`) passed to the upBeat API was NULL.

EIO

The application encountered a problem when trying to send a message to the upBeat daemon.

EMSGSIZE

The upBeat daemon is sending a packet that is too big for the application to handle. It is likely that EPROTONOSUPPORT would be seen first, allowing the application to avoid EMSGSIZE.

ENOTCONN

A connection to the local upBeat daemon is maintained by `libupbeat` on behalf of the application. When ENOTCONN is returned, it indicates that `libupbeat` has lost this connection to the upBeat daemon and cannot re-establish it. When returned by `ubInit()`, this code indicates that a connection was never made.

EPROTO

A packet received from the upBeat daemon contained a protocol error.

EPROTONOSUPPORT

There is a version conflict. The application is attempting (unsuccessfully) to interact with an incompatible upBeat daemon.

6 upBeat Sample Applications

This chapter explains the three sample applications included with your upSuite HA software: `wrapper`, `status`, and `mt_multi_svc`. Each of these applications demonstrates an aspect of upBeat's functionality. `wrapper` demonstrates initialization and failover, `status` performs an ongoing system status checkup, and `mt_multi_svc` handles multiple threads and split brain events. These programs use the upBeat APIs and are intended to provide examples you can follow when developing your own client applications.

The source code for each program is located in `/opt/upsuite/src/upbeat`.

In this chapter

The `wrapper` Sample Application 57
The `status` Sample Application 64
The `mt_multi_svc` Sample Application 69

The **`wrapper`** Sample Application

`wrapper` demonstrates the major initializing and failover functions of upBeat. `wrapper` is a proxy service; the name of a service is passed to `wrapper` on its command line. The service passed in to `wrapper` must be configured in `upsuite.conf`.

`wrapper` first calls `ubInit()` and other commands to learn the state of the network. `wrapper` then registers the service that was passed in, requesting active status. It then waits for upBeat to designate the service as standby or active. If upBeat directs the service to be standby, `wrapper` waits indefinitely. However, if upBeat directs the service to become active, `wrapper` then runs the given command (in the example later in this section, **`iostat 10`** is used) and waits for it to finish.

After the command has finished, `wrapper` registers the service as standby. upBeat then fails the service over to an available server. If there is no server available, upBeat will make the service active again on the same server.

The source code for `wrapper` is located in `wrapper.c`.

To Use **wrapper**

Step	Description	Command
1.	Change directory.	**cd**
2.	Copy the directory `src/upbeat`.	**cp -r /opt/upsuite/src/upbeat ./myupbeat**
3.	Go the `myupbeat` directory.	**cd myupbeat**
4.	Build using the supplied make file.	**make**
5.	Run `wrapper` on both servers. Press **`Ctrl-C`** on the active side to cause a failover. On the new active side, press **`Ctrl-C`** to failback. Note: To best demonstrate upBeat failover, use a command that does not return immediately (e.g., **`iostat 10`**). This way, your program will run the command every so many seconds. Note: To stop `wrapper` you will need to press **`Ctrl-C`** *twice*. The first stops the command; the second stops the program.	**wrapper *<service>* *<command>***
	`left#` **`wrapper upstart "iostat 10"`**	

Source Code of `wrapper`

```
#include <stdio.h>
#include <errno.h>
#include <poll.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <libupbeat.h>
```

```
#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

static void directive(void*, ub_svc_id_t, ub_svc_status_t);

static ub_svc_id_t glob_sid;
static ub_node_id_t glob_id, glob_peerid;
static char *glob_prog;

int
main(int argc, char *argv[])
{
    upbeat_t        *ubp = NULL;
    char            *svcname;
    ub_service_ip_t service_ip, *sip = &service_ip;
    ub_ippair_t     ippair, *ippp = &ippair;

    /*
     * These are the callbacks we will pass to ubGetState() and
     * ubAsync(). UpBeat passes ubp as an argument to the callbacks.
     * We do not have node, link, and service callbacks; we do have
     * a directive callback, so we can negotiate active or standby.
     */
    ub_ops_t ub_ops = {
        ubp,    /* not set yet */
        NULL,   /* no node callback */
        NULL,   /* no link callback */
        NULL,   /* no service callback */
        directive
    };
    ub_ops_t            *opsp = &ub_ops;

    /*
     * Check command line arguments.
     *
     * Note that argv[2] is the entire command to run,
     * and it may need to be quoted in the shell if it
     * has arguments. Example:
     *
```

```
 *    wrapper <svcname> "iostat 10"
 */
if (argc != 3) {
    fprintf(stderr, "Usage: %s <servicename> <command>\n", argv[0]);
    exit (1);
}
svcname = argv[1];
glob_prog = argv[2];

if ((ubp = ubInit()) == NULL) {
    fprintf(stderr, "ubInit() failed: errno %d %s\n",
      errno, strerror(errno));
    exit (1);
}
ub_ops.arg = (void *)ubp;
printf("upbeat is running...\n");

/*
 * Find our node in the upSuite configuration.
 */
if ((glob_id = ubNode(ubp, NULL))) {
    printf("we are node %d\n", glob_id);
} else {
    fprintf(stderr, "upsuite misconfiguration: this system does not "
                    "match any node in the upSuite configuration.\n");
    exit (1);
}

/*
 * Find our service in the upSuite configuration.
 */
if ((glob_sid = ubSvc(ubp, svcname))) {
    printf("service id = %d; service name = %s\n", glob_sid, svcname);
} else {
    fprintf(stderr, "no such service: %s\n", svcname);
    exit (1);
}

/* peerid may be 0 */
glob_peerid = ubSvcPeer(ubp, glob_id, glob_sid);
printf("service peer is: %d\n", glob_peerid);

/*
 * Get service IP addresses -- the addresses upbeat manages
 * during IP failover.
```

```
 *
 * Note: this application does not use these addresses; this
 * code is include for the purposes of illustration.
 */
sip->addr = 0;
while (ubServiceIP(ubp, glob_sid, sip->addr, sip)) {
    printf("service ip: %s %s\n", sip->interface, sip->ip);
}

/*
 * Get pairs of IP addresses the active and standby can use
 * to communicate.
 *
 * Note: this application does not use these addresses; this
 * code is include for the purposes of illustration. If an
 * application did use these addresses, it would also want to
 * register a link callback.
 */
ippp->local = 0;
while (ubSvcIPPair(ubp, glob_id, glob_sid, ippp->local, ippp)) {
    struct sockaddr_in  sockaddr_in;

    sockaddr_in.sin_addr.s_addr = ippp->local;
    printf("local = %s, ", inet_ntoa(sockaddr_in.sin_addr));
    sockaddr_in.sin_addr.s_addr = ippp->remote;
    printf("remote = %s\n", inet_ntoa(sockaddr_in.sin_addr));
}

/*
 * Give libupbeat a chance to do housekeeping.
 */
if (ubGetState(ubp, opsp) == -1) {
    fprintf(stderr, "ubGetState() failed: errno %d %s\n",
      errno, strerror(errno));
    exit (1);
}

/*
 * Register to provide our service. We register as willing
 * to be active, but there is no guarantee we will be told
 * to be active.
 */
```

```
    if (ubRegSvc(ubp, glob_sid, UB_ACTIVE) == -1) {
        fprintf(stderr, "ubRegSvc() failed: errno %d %s\n",
          errno, strerror(errno));
        exit (1);
    }

    while (1) {
        struct pollfd    pollfd, *pfdp = &pollfd;
        int              poll_timeout_msec = 500;

        /*
         * Fill in the pollfd fresh each time. upBeat will reestablish
         * a connection if it is dropped, so the fd may change.
         */
        ubSetupPollfd(ubp, pfdp);

        /*
         * We are only polling for upBeat, but we could be polling
         * for any number of file descriptors.
         */
        if (poll(pfdp, 1, poll_timeout_msec) == -1 ) {
            fprintf(stderr, "poll() failed: %d %s\n", errno, strerror(errno));
            exit (1);
        }

        /*
         * We could check the return value from poll() and the contents
         * of pollfd to see if we have any events, but ubAsync() checks
         * again anyway, plus it gets a chance to do some housekeeping.
         */
        if (ubAsync(ubp, opsp) == -1) {
            fprintf(stderr, "ubAsync() failed: errno %d %s\n",
              errno, strerror(errno));
            exit (1);
        }
    }

    /* NOTREACHED */
    exit (0);
}

/*
 * This application blocks when we call system(glob_prog) until
 * glob_prog completes, which means we will not call ubAsync()
 * or anything else for the duration.
```

```
 *
 * A multithreaded application (or an application that forked, execed,
 * and managed the child process) could continue to poll, and could
 * respond to UB_STANDBY message by killing the child process.
 */
static void
directive(void *arg, ub_svc_id_t service, ub_svc_status_t status)
{
    static int    rereg = FALSE;
    upbeat_t      *ubp = (upbeat_t *)arg;

    switch (status) {
    case UB_ACTIVE:
        printf("directive: UB_ACTIVE\n");
        if (ubAckSvc(ubp, glob_sid, status) == -1) {
            fprintf(stderr, "ubAckSvc() failed: errno %d %s\n",
              errno, strerror(errno));
            exit (1);
        }
        break;
    case UB_STANDBY:
        printf("directive: UB_STANDBY\n");
        if (ubAckSvc(ubp, glob_sid, status) == -1) {
            fprintf(stderr, "ubAckSvc() failed: errno %d %s\n",
              errno, strerror(errno));
            exit (1);
        }
        if (rereg) {
            /*
             * Give the standby a chance to beat us in. From an
             * HA perspective this is not really necessary if we
             * really are ready to run again, but for demonstration
             * purposes it forces a failover.
             */
            sleep(3);

            if (ubRegSvc(ubp, glob_sid, UB_ACTIVE) == -1) {
                fprintf(stderr, "ubRegSvc() failed: errno %d %s\n",
                  errno, strerror(errno));
                exit (1);
            }
            rereg = FALSE;
        }
        break;
```

```
        default:
            printf("bad directive: %d\n", status);
            exit (1);
            break;
        }

        if (status == UB_ACTIVE) {
            system(glob_prog);

            /* Tell upBeat we are no longer serving. */
            if (ubRegSvc(ubp, glob_sid, UB_STANDBY) == -1) {
                fprintf(stderr, "ubRegSvc() failed: errno %d %s\n",
                  errno, strerror(errno));
                exit (1);
            }

            /* Once we have acknowledged standby, reregister for active. */
            rereg = TRUE;
        }

        return;
    }
```

The `status` Sample Application

`status` demonstrates upBeat's installation verification process. Its output notifies you of all available links, nodes, and services of which upBeat is aware. However, `status` does not offer any services.

The source code for `status` is located in `status.c`.

To Use `status`

Running `status` with the **`-q`** flag causes `status` to print upBeat's full node, link, and service status once, then exit. Running it without the **`-q`** flag causes `status` to print upBeat's full node, link, and service status once, then run indefinitely, printing any changes to status. To stop `sample`, use **`Ctrl-C`**.

Step	Description	Command
1.	Change directory.	**`cd`**
2.	Copy the directory `src/upbeat`.	**`cp -r /opt/upsuite/src/upbeat ./myupbeat`**

Step	Description	Command
3.	Go the `myupbeat` directory.	**`cd myupbeat`**
4.	Make a file.	**`make`**
5.	Run `status`. Note: To stop `sample` press **`Ctrl-C`**.	**`status`**
	`left# status`	

Source Code of `status`

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <poll.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <libupbeat.h>

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

static void unode(void*, uint32_t, ub_status_t, boolean_t);
static void ulink(void*, uint32_t, ub_status_t, boolean_t);
static void uservice(void*, uint32_t, uint32_t, ub_status_t, boolean_t);
static void usage(char*);

int
main(int argc, char *argv[])
{
    ub_node_id_t    node_id;
    upbeat_t        *ubp = NULL;
    int             c;
```

```
int             quick = FALSE;
extern int      optind;

/*
 * These are the callbacks we will pass to ubGetState() and
 * ubAsync(). UpBeat passes ubp as an argument to the callbacks.
 * We have node, link, and service callbacks for status; we do not
 * have a directive callback, since we are not offering a service,
 * and so will not be active or standby.
 */
ub_ops_t ub_ops = {
    ubp, /* not set yet */
    unode,
    ulink,
    uservice,
    NULL /* no directive callback */
};
ub_ops_t *opsp = &ub_ops;

/*
 * Check command line arguments.
 *
 *    -q tells us to run ubGetState() once and not run ubAsync() at all.
 *
 *    -h or -? print the usage line.
 */
while ((c = getopt(argc, argv, "hq")) != EOF) {
    switch (c) {
    case 'q':
        /* Check status, then exit; do not loop. */
        quick = TRUE;
        break;
    default:
    case 'h':
    case '?':
        usage(argv[0]);
        break;
    }
}
if (optind != argc) usage(argv[0]);

/*
 * Initialize libupbeat and the connection to the upbeat daemon.
 */
```

```
if ((ubp = ubInit()) == NULL) {
    fprintf(stderr, "ubInit() failed: errno %d %s\n",
      errno, strerror(errno));
    exit (1);
}
ub_ops.arg = (void *)ubp;
printf("upbeat is running...\n");

/*
 * Find our node in the upSuite configuration.
 */
if ((node_id = ubNode(ubp, NULL)) == 0) {
     printf("this system does not match "
            "any node in the upSuite configuration.\n");
} else {
    printf("this system is node %d\n", node_id);
}

/*
 * Get the current state. Any changes to this state will be reported
 * afterward via ubAsync().
 */
if (ubGetState(ubp, opsp) == -1) {
    fprintf(stderr, "ubGetState() failed: errno %d %s\n",
      errno, strerror(errno));
    exit (1);
}

while (! quick) {
    struct pollfd    pollfd, *pfdp = &pollfd;
    int              poll_timeout_msec = 500;

    /*
     * Fill in the pollfd fresh each time. libupbeat will reestablish
     * a connection if it is dropped, so the fd may change.
     */
    ubSetupPollfd(ubp, pfdp);

    /*
     * We are only polling for upBeat, but we could be polling
     * for any number of file descriptors.
     */
```

```
        if (poll(pfdp, 1, poll_timeout_msec) == -1 ) {
            fprintf(stderr, "poll() failed: %d %s\n", errno, strerror(errno));
            exit (1);
        }

        /*
         * We could check the return value from poll() and the contents
         * of pollfd to see if we have any events, but ubAsync() checks
         * again anyway, plus it gets a chance to do some housekeeping.
         */
        if (ubAsync(ubp, opsp) == -1) {
        fprintf(stderr, "ubAsync() failed: errno %d %s\n",
          errno, strerror(errno));
            exit (1);
        }
    }

    ubFini(ubp);

    exit (0);
}

static void
unode(void *arg, uint32_t id, ub_status_t status, boolean_t async)
{
    printf("node=%d, status=%s, %s\n",
      id,
      status == UB_STATUS_UP ? "UP" : "DOWN",
      async ? "ASYNC" : "SYNC"
    );

    return;
}

static void
ulink(void *arg, uint32_t id, ub_status_t status, boolean_t async)
{
    struct    in_addr inaddr;

    inaddr.s_addr = id;
```

```
    printf("\tlink=%s, status=%s, %s\n",
      inet_ntoa(inaddr),
      status == UB_STATUS_UP ? "UP" : "DOWN",
      async ? "ASYNC" : "SYNC"
    );

    return;
}

static void
uservice(void *arg, uint32_t sid, uint32_t id, ub_status_t status, boolean_t async)
{
    printf("service=%d, node=%d, status=%s, %s\n",
      sid,
      id,
      status == UB_STATUS_UP ? "UP" : "DOWN",
      async ? "ASYNC" : "SYNC"
    );

    return;
}

static void
usage(char *progname)
{
    fprintf(stderr, "Usage: %s [-q(uick)]\n", progname);

    exit (1);

    /* NOTREACHED */
    return;
}
```

The mt_multi_svc Sample Application

The mt_multi_svc application is a multi-threaded program managing split brain events and multiple services via sockets. Interaction with the application is accomplished via a telnet client.

The mt_multi_svc application has three primary threads: the main thread, the upBeat Interface Thread, and the Socket Acceptance Thread. The upBeat Interface Thread interacts with the upBeat daemon and detects and handles the split brain condition. The Socket Acceptance Thread accepts clients for services and spawns other threads to handle those services.

Upon instantiation, the application becomes a daemon and the main thread instantiates the other threads. The main thread then acts as a signal handler, receiving the `SIGUSR1` signal from the upBeat Interface Thread upon a change of service state, for example, a service going from standby to active or vice versa. Concurrency is achieved among the other threads via pipes, condition variables, and mutexes. You can terminate the application either through one of the services or by sending a `SIGTERM` or `SIGQUIT` signal to the application via the `kill` command at the console.

This section explains the following about the `mt_multi_svc` application:

- Configuration File
- System Architecture
- Application Architecture
- Termination
- To Use `mt_multi_svc`

The source code is located in `/opt/upsuite/src/upbeat/mt_multi_svc.c`.

Configuration File

For your reference, below is an example of a configuration file (`/etc/upsuite.conf`) you could use on each of the two systems using `mt_multi_svc`. Your configuration, however, may require a different setup.

```
<?xml version="1.0" ?>

<UpSuiteConfig VERSION="2">

<UPBEAT STARTUPDELAY_SEC="5"/>

<NETWORK NAME="Network1" DESCRIPTION="172.17.33.0/24"/>
<NETWORK NAME="Network2" DESCRIPTION="172.18.33.0/24"/>

<NODE NAME="left" NODE_ID="1" DESCRIPTION="SPARC CP1500 Solaris 2.8">
   <INTERFACE NAME="hme0" NETWORK="Network1" IP="172.17.33.120"/>
   <INTERFACE NAME="hme1" NETWORK="Network2" IP="172.18.33.120"/>
</NODE>

<NODE NAME="right" NODE_ID="2" DESCRIPTION="SPARC CP1500 Solaris 2.8">
   <INTERFACE NAME="hme0" NETWORK="Network1" IP="172.17.33.121"/>
   <INTERFACE NAME="hme1" NETWORK="Network2" IP="172.18.33.121"/>
</NODE>
```

```
<HEARTBEAT
   NAME="left -- right"
   TYPE="POINT_TO_POINT"
   TIMEOUT_MSEC="2000"
   RESEND_MSEC="650">
   <NODE_REF NODE_ID="1"/>
   <NODE_REF NODE_ID="2"/>
   <LINK NETWORK="Network1"/>
   <LINK NETWORK="Network2"/>
</HEARTBEAT>

<SERVICE
   NAME="MtAppCtl"
   SERVICE_ID="50"
   TYPE="BASIC"
   STARTUPDELAY_SEC="5"
   PORT="1800">
   <NODE_REF NODE_ID="1"/>
   <NODE_REF NODE_ID="2"/>
   <LINK NETWORK="Network1"/>
   <LINK NETWORK="Network2"/>
   <SERVICE_IP IP="192.168.1.1" IF="hme0:20"/>
   <SERVICE_IP IP="192.168.1.2" IF="hme0:21"/>
   <SERVICE_IP IP="192.168.1.3" IF="hme0:22"/>
</SERVICE>

<SERVICE
   NAME="MtSysCmd"
   SERVICE_ID="51"
   TYPE="BASIC"
   STARTUPDELAY_SEC="5"
   PORT="1801">
   <NODE_REF NODE_ID="1"/>
   <NODE_REF NODE_ID="2"/>
   <LINK NETWORK="Network1"/>
   <LINK NETWORK="Network2"/>
   <SERVICE_IP IP="192.168.1.4" IF="hme0:23"/>
   <SERVICE_IP IP="192.168.1.5" IF="hme0:24"/>
</SERVICE>

</UpSuiteConfig>
```

System Architecture

The architecture is that of an HA system, with two nodes configured identically with the same software components on each. One node currently contains the services in the standby state, while the other contains the services in the active state.

The application provides two services, with each service on different “floating” IP addresses and ports. A CCN control node can be used to control the nodes (rebooting and bringing links up and down), to start and stop the application, and to view the generated log file. One or more telnet client machines are used to communicate with the services on either system.

`mt_multi_svc` creates a log file in the `/tmp/mt_multi_svc` directory on each system where application activity is recorded.

The following illustration shows the deployment of `mt_multi_svc`:

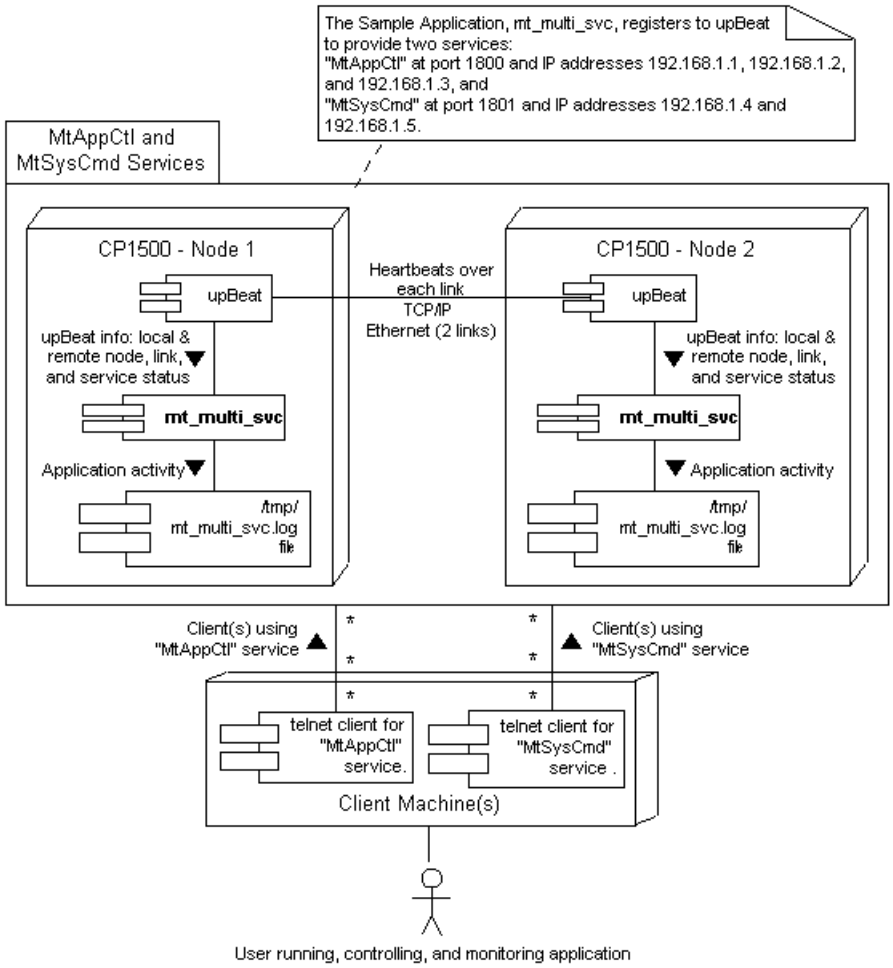
The Sample Application, mt_multi_svc, registers to upBeat to provide two services:
"MtAppCtl" at port 1800 and IP addresses 192.168.1.1, 192.168.1.2, and 192.168.1.3, and
"MtSysCmd" at port 1801 and IP addresses 192.168.1.4 and 192.168.1.5.
MtAppCtl and MtSysCmd Services
CP1500 - Node 1
upBeat
upBeat info: local & remote node, link, and service status
mt_multi_svc
Application activity
/tmp/ mt_multi_svc.log file
Heartbeats over each link
TCP/IP
Ethernet (2 links)
CP1500 - Node 2
upBeat
upBeat info: local & remote node, link, and service status
mt_multi_svc
Application activity
/tmp/ mt_multi_svc.log file
Client(s) using "MtAppCtl" service
Client(s) using "MtSysCmd" service
telnet client for "MtAppCtl" service.
telnet client for "MtSysCmd" service .
Client Machine(s)
User running, controlling, and monitoring application

Figure 3 `mt_multi_svc` system architecture

Application Architecture

The following illustration is a component diagram illustrating `mt_multi_svc`'s architecture.

Notice that `mt_multi_svc` first becomes a daemon process. It is composed of multiple threads, specifically:

- **The Main Thread**
  This thread is responsible for spawning the upBeat Interface Thread and the Socket Acceptance Thread. The main thread acts as the signal handler for `mt_multi_svc`.
- **The upBeat Interface Thread**
  This thread is responsible for interfacing to upBeat. The upBeat Interface Thread is also responsible for handling the split brain condition.
- **The Socket Acceptance Thread**
  This thread is responsible for accepting clients for both services, and then spawning other threads that actually handle the service.

All threads have access to the log file.

If `mt_multi_svc` is used as a model for your own applications, you can substitute your services for those of `mt_multi_svc`.

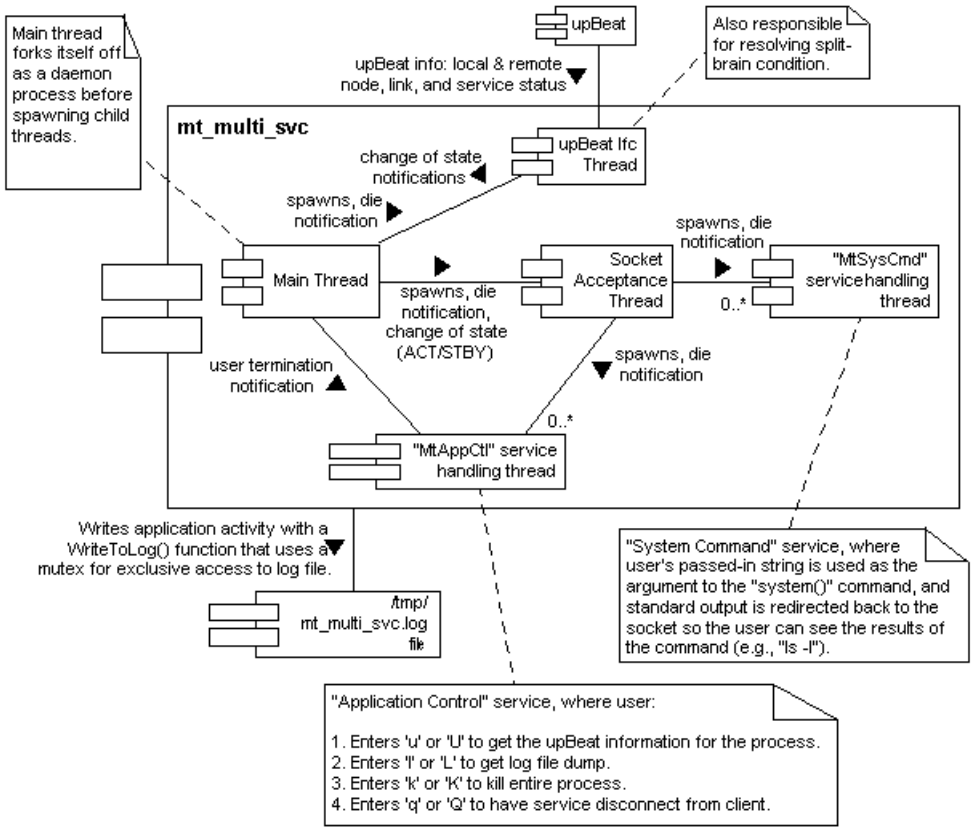
Main thread forks itself off as a daemon process before spawning child threads.
upBeat
Also responsible for resolving split-brain condition.
upBeat info: local & remote node, link, and service status
mt_multi_svc
upBeat Ifc Thread
change of state notifications
spawns, die notification
spawns, die notification
Main Thread
Socket Acceptance Thread
"MtSysCmd" servicehandling thread
0..*
spawns, die notification, change of state (ACT/STBY)
user termination notification
spawns, die notification
0..*
"MtAppCtl" service handling thread
Writes application activity with a WriteToLog() function that uses a mutex for exclusive access to log file.
/tmp/ mt_multi_svc.log file
"System Command" service, where user's passed-in string is used as the argument to the "system()" command, and standard output is redirected back to the socket so the user can see the results of the command (e.g., "ls -l").
"Application Control" service, where user:
1. Enters 'u' or 'U' to get the upBeat information for the process.
2. Enters 'l' or 'L' to get log file dump.
3. Enters 'k' or 'K' to kill entire process.
4. Enters 'q' or 'Q' to have service disconnect from client.

Figure 4 mt_multi_svc application architecture

Termination

You can terminate the `mt_multi_svc` application at the console by using the `kill` utility to send a `SIGTERM` or `SIGQUIT` signal to the application.

A user connected to the Application Control service (`MtAppCtl`) can terminate the application by issuing a `kill` command to the service. The command is sent over a socket to an `MtAppCtl` service handling thread which, after a positive confirmation, then issues a `SIGQUIT` signal to the main thread.

To Use mt_multi_svc

To use mt_multi_svc, invoke the program on both of your systems by performing the following steps:

Step	Description	Command
1.	Change directory.	**cd**
2.	Copy the directory src/upbeat.	**cp -r /opt/upsuite/src/upbeat ./myupbeat**
3.	Change to the myupbeat directory.	**cd myupbeat**
4.	Build using the supplied makefile.	**make**

Step	Description	Command
5.	Run `mt_multi_svc`. The command's options are explained below.	**`mt_multi_svc [-s method] [-h]`**

Command usage

`mt_multi_svc [-s method] [-h]`

Command options

Including the **`-h`** flag causes the program to print its usage information and then exit.

Including the **`-s`** flag denotes split brain handling where ***`method`*** is one of the following case-insensitive options:

- **`node1`** (default behavior if no method is specified)
  Using this method causes the program to keep the services of the node with the lower node ID active if a split brain is detected. The services of the other node will become standby. The node with the lower ID does not literally have to be named node1.
- **`node2`**
  Using this method causes the program to keep the services of the node with the higher node ID active if a split brain is detected. The services of the other node will become standby. The node with the higher ID does not literally have to be named node2.
- **`firstactive`**
  Using this method causes the program to keep the services of the node whose services were active first upon program instantiation to become or remain active if a split brain is detected. The services of the other node will become standby.
- **`lastactive`**
  Using this method causes the program to keep the services of the node whose services were standby first upon program instantiation to become or remain active if a split brain is detected. The services of the other node will become standby.

Part II: upDisk

upDisk is a software module that provides simple, reliable file system replication over a network. By relying on redundant systems, storage, and networks, upDisk allows system configurations that prevent any single point of failure. upDisk continually replicates data over a redundant network to a standby file system.

upDisk is used as part of upSuite HA™ and is available for Solaris 7, 8, 9 and 10 on SPARC.

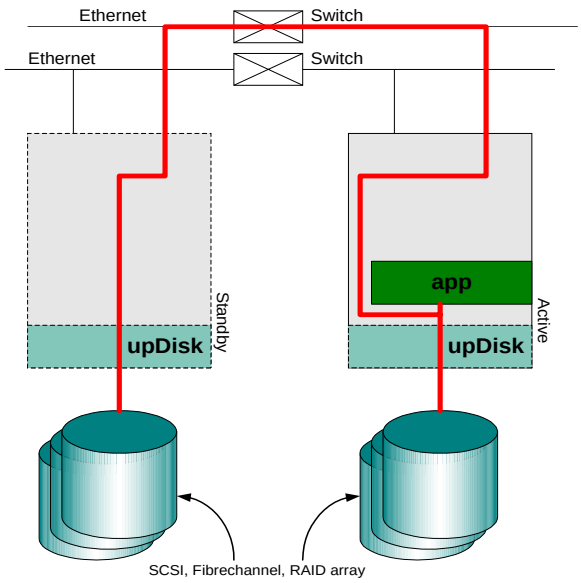

Figure 1 Example upDisk configuration

7 Introduction to upDisk

This chapter introduces you to upDisk, a software module used to replicate file system data as part of upSuite HA™, a series of software modules designed to provide transparent high-availability capability for applications.

In this chapter

What is upDisk? 81
Why Should I Use upDisk? 81
How Does upDisk Work? 82

What is upDisk?

upDisk provides simple, reliable file system replication over a network. upDisk performs all of the functions necessary for completing simultaneous disk writes across a TCP/IP network at wireline speeds.

Features of upDisk include:

- Simple administration
- No disruption during failover
- Rapid recovery from temporary network failures
- Automatic or manual recovery from prolonged outages or equipment replacement
- Full integration with upBeat™ architecture
- Easy HA NFS construction
- No API or application changes necessary

The upDisk server software and optional client API are available for Solaris/SPARC 7, 8, 9 and 10.

Why Should I Use upDisk?

By relying on redundant systems, storage, and networks, upDisk allows system configurations that prevent any single point of failure. This fact significantly reduces the need for single point hardening and creates a new level of flexibility, dramatically lowering the cost of redundant application and NFS server systems deployment.

In addition, other benefits of upDisk include:

- Control over which files are in replicated directories and which files are in conventional directories
- Freedom from worry about application data redundancy
- High availability provided to all NFS clients

How Does upDisk Work?

upDisk continually replicates data over a redundant network to a standby file system as illustrated in Figure 1. upDisk replicates file system operations so that those occurring on the active happen simultaneously on the standby. Only operations that change the file system are sent across the network. For example, reads happen on the active only, but creates and writes are sent to the standby as well. Note that while the file systems are referred to as “Active” and “Standby,” activity is constantly occurring on both machines.

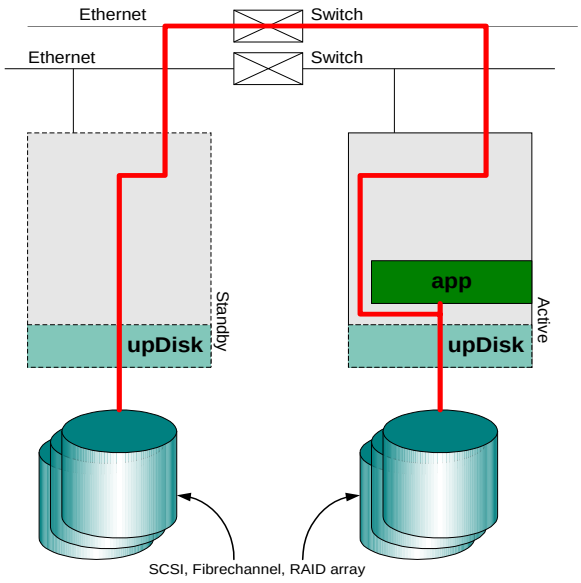

Figure 1 Example upDisk configuration

As you can see in the example above, the active processor writes information to its local disk while that information is simultaneously written to the disk of a standby processor via the network. The specific disk media, be it standard SCSI, Fibre channel, or RAID arrays, is irrele-

vant to upDisk. Note that the standby file systems are read-only for processes on the standby side.

Because upDisk replicates file systems, the replicated file systems may be physically located on two separate servers. Therefore, server A may be active on file system 1, and standby on file system 2; server B may be standby on file system 1 and active on file system 2. This configuration is illustrated in Figure 2.

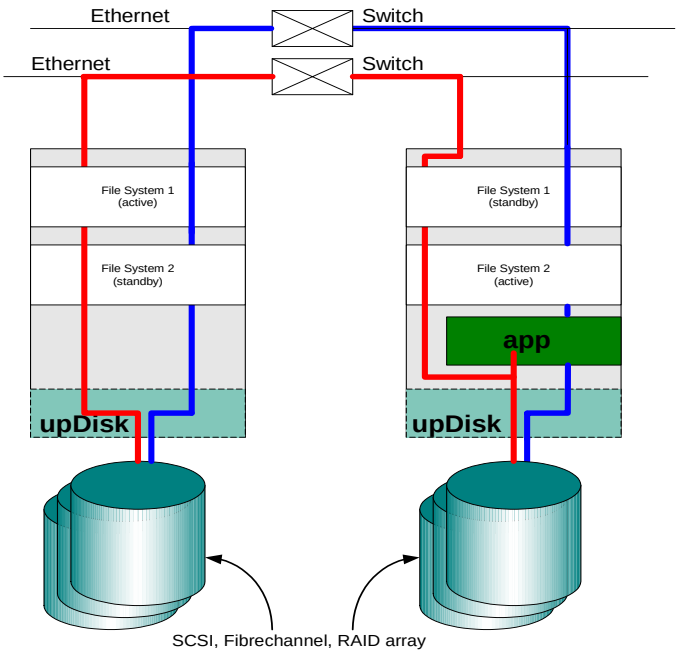

Figure 2 Multiple file system replication operation

Tracing the route that a disk write takes through the system helps to illuminate how upDisk executes its tasks as illustrated in Figure 3. The typical system call, a `write()` in this case, is sent from the application to the kernel. The call then passes to the VFS (virtual file system) before going to the local file system, for example, UFS (UNIX file system). At this point, the call becomes a system-specific call to a device driver for the storage medium in question.

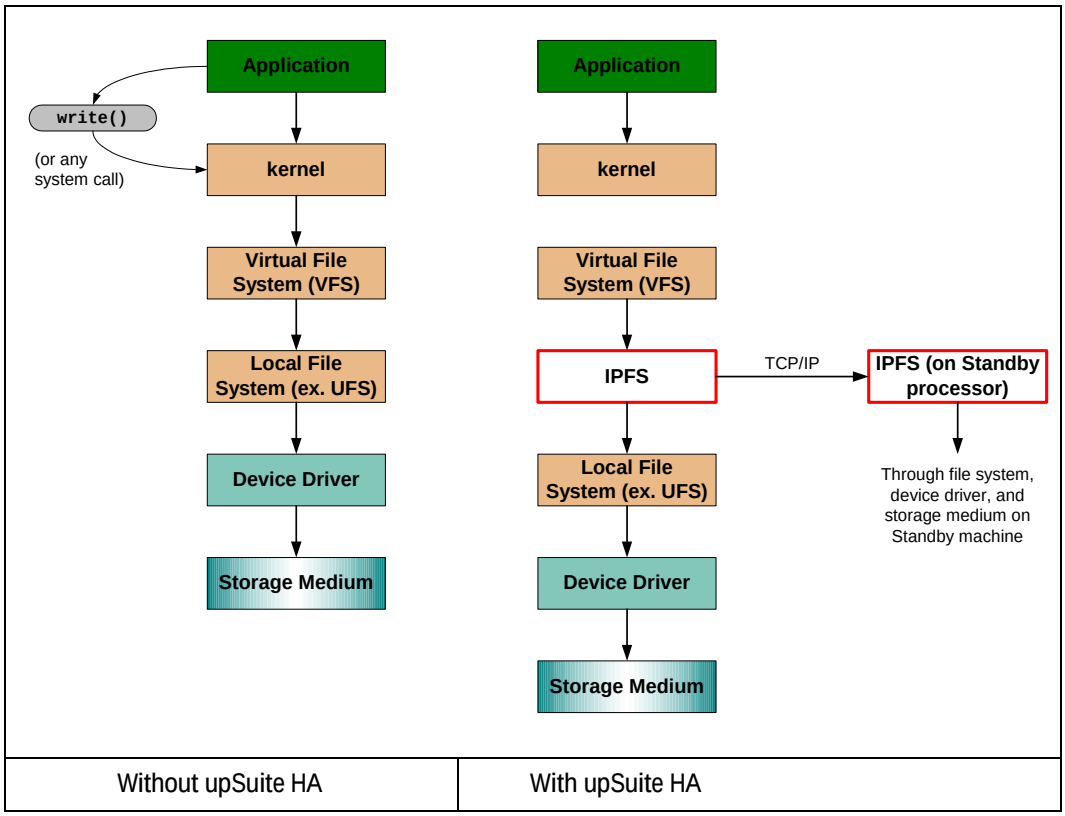

Figure 3 Write execution

upDisk accomplishes simultaneous writes by inserting an extra layer in this process. Where upDisk has been installed, a virtual file system called `ipfs` exists between the VFS and the local FS (file system) layer. The system call is passed from the VFS to `ipfs`. `ipfs` then passes the call both to the local FS and also laterally over the network, to another instance of `ipfs` on a second machine. This instance of `ipfs` passes the write down through the standby processor's local file system to its disk array.

This implementation has several advantages. As a file system, `ipfs` is transparent upwards and downwards. The application makes the same system calls, which are intercepted at the VFS layer and propagated over the network link. Below `ipfs`, the local file system and disk drivers are not affected, so any advantages in redundancy or performance inherent in the local file system or disk subsystem are preserved.

`write` operations offer an interesting example of `updisk`/`ipfs` settings.

Below is the typical sequence of an application `write`:

1. An application calls a `write()`.
2. The `write()` system call is vectored to `ipfs`.
3. `ipfs` does the local FS `write`.
4. `ipfs` sends the operation over the network to the standby `ipfs` (after the local FS operation completes).
5. TCP acknowledges the transmission of the operation and `ipfs` sends a receive acknowledgement.
6. The standby `ipfs` does the standby local FS `write`.
7. The standby `ipfs` sends an acknowledgement over the network to the active `ipfs` (after the standby local FS operation completes).

In non-upDisk systems, file system permissions and open flags control `write()` access to a file. `ipfs`, however, will allow a file to be open for writing on the standby but will not permit a write to succeed on that machine until it is the active. This means that applications can open a file for writing in order to become operational, but will not be permitted to write to the file until the standby system becomes the active node. Files on the active system are always available for writing.

Writes for FIFOs and devices (special files) are not replicated because they are local in scope. As such, writes to these files are allowed on both the active and standby at all times.

Replay and Repair

In normal operation, upDisk sends any file system operations over the network to the standby server. If the standby system is unavailable, the active server tracks changes. When the standby server is once again available, the active server sends over all missing changes. This function is called “replay.”

If upDisk ever detects that the two servers are out of sync, or that either local file system has been damaged, it performs a complete checksum on both sides and makes necessary changes. This function is called “repair.” If a repair fails for any reason, upDisk tries to repair again until the repair is successfully completed.

8 upDisk Administration

In this chapter we detail issues specific to system administrators. When applied to upDisk, common software practices may have unexpected results of which system administrators should be apprised.

upDisk Administrator Considerations

This section describes some situations which should be handled with care.

Stopping the Solaris machine

There are several ways to shut down a Solaris system, and not all of them ensure a graceful exit for upSuite software. The following shutdown techniques run the upSuite shutdown scripts, and are therefore recommended when you need to stop a Solaris system where upSuite is running:

- **`/usr/sbin/shutdown`** (man `shutdown`)
- **`/sbin/init`** (man `init`)

The following shutdown methods cause a repair, and should therefore be avoided if possible: **`/usr/sbin/reboot`**, **`/usr/sbin/halt`**, and **`/usr/ucb/shutdown`**.

To reboot the system, the method we recommend to ensure proper upSuite operation is to use the following command:

```
/usr/sbin/shutdown -i 6 -g 0
```

Stopping the NFS server

If you run the command **`/etc/init.d/nfs.server stop`**, NFS turns off and nothing is shared. If you then run **`nfs.server start`**, NFS will turn on if there are entries in `/etc/dfs/dfstab`; however, no upDisk directories will be shared whether or not NFS is turned on. Therefore, do not run **`nfs.server stop`**. There is no harm in running **`nfs.server start`**, even if NFS is already running. If you need to stop the Solaris NFS server manually with **`nfs.server stop`**, you must then stop and restart upDisk to re-share the HA NFS datasets. The sequence of events and commands would be as follows:

1. The following command stops the Solaris NFS server and unshares all `dfstab` and HANFS shares:
   `/etc/init.d/nfs.server stop`

2. The following command restarts the Solaris NFS server:
   `/etc/init.d/nfs.server start`
   At this point, HANFS shares are not shared.

3. The following commands stop and restart upDisk:
   `/etc/init.d/updisk stop`
   `/etc/init.d/updisk start`
   After these commands run, HANFS shares reappear

Manipulating underlying file systems

Do not manipulate an underlying file system or subdirectory in any way. Doing so will, at the very least, result in all system changes going unreplicated; at the very worst, both servers will shut down.

Shutting down the system when a link is up

upDisk file systems cannot be unmounted if the pair of servers has an established link. Under some circumstances, this can cause the system to hang during a shutdown.

Normally, the system calls `/etc/init.d/updisk stop` as part of the shutdown sequence. However, if you run either **`/usr/ucb/shutdown`** or **`init 0`** to shut down the system, be aware that these cause the system to initiate single-user mode without going through the full shutdown sequence. To avoid this problem, use one of the following to shut down your system:

`halt`

`/etc/shutdown`

`/usr/ucb/shutdown -h`

`init -s`

`init -S`

Monitoring upDisk

You can monitor the file system with the **udstat** command. See “udstat” on page 132 for further explanation and usage of this command.

On Solaris 8, you can also get `ipfs` statistics using the `kstat` command. On Solaris 7, the `kstat ioctl(2)` interface is supported, but the `kstat(1M)` command does not exist.

Logs

upDisk’s status can be monitored via the `syslog`. Typically, all `ipfs` messages are routed to `/var/adm/messages`. upDisk daemon messages are typically routed to `/var/log/upsuite`. Critical messages are automatically routed to your console.

The file `/var/log/nidb` contains error information from an upDisk component, a program called the nidb daemon, which is used for file handle translation.

Troubleshooting Console Messages

This section discusses console messages you might see that indicate some common issues that arise when using upDisk and tells how to respond to these events.

Bringing Systems Online

When `ipfs` is first provided a link for use, internal `ipfs` consistency checks are made between the two systems. If these checks do not turn up any problems, or turn up problems that are not urgent, `ipfs` will continue running. Otherwise, `ipfs` drops the link and informs upDisk of the problem.

In at least one case, the warning message may only indicate that `ipfs` does not have all the information upDisk does about a given situation.

You may encounter one or more of the following messages upon bringing your systems up.

Problems with startup exchange

```
Oct  3 09:52:41 left unix: /ipfs: GDAY failed - losing link
```

The message above indicates that a network error occurred during the initial exchange; upDisk will attempt to restore the connection immediately. The network problems should be addressed, but upDisk will continue to provide links while able.

```
Oct  3 09:52:41 left unix: /ipfs: GDAY wrong - losing link
```

The message above indicates that the initial exchange was invalid or corrupt. upDisk will

attempt to restart services. If the problem continues, it generally indicates that another application is using the remote port and upDisk is not running on that system.

Operator intervention required

```
Oct  3 09:52:41 left unix: /ipfs: role link state - dropping
   link, operator intervention for (split) required
```

The message above indicates that a condition requiring operator intervention has previously been detected, and `ipfs` cannot proceed with operations until the problem has been fixed. The link is dropped and upDisk notes the reason required for operator intervention in status displays.

Active/Standby Mismatches

```
Oct  3 09:52:41 left unix: ROLE CLASH - local ACTIVE remote
   ACTIVE
Oct  3 09:52:41 left unix: ROLE CLASH - local STANDBY remote
   STANDBY
```

The messages above indicate that both systems are either active or standby and `ipfs` cannot proceed. upDisk will assume a split brain condition.

Operator intervention is required before operations will proceed.

Warning messages about the underlying file system

```
Oct  3 09:52:41 left unix: DIRTY detected - active SYNC
   standby DIRTY
```

The message above indicates `ipfs` has detected that an underlying file system is “dirty” (in need of repair) and that `ipfs` expected to be in repair mode, but is not. In other words, if either of the file systems are dirty, and `ipfs` is not in repair, then the above message is generated.

Note, however, that there is one circumstance under which you may be sent the above message unnecessarily. If a link drops during replay, upDisk restarts the replay again. `ipfs` does not have enough information to know that this is the case and thus warns that the standby is dirty.

In each of the cases described above, no action is required.

```
Oct  3 09:52:41 left unix: NEW FS detected - active OLD standby NEW
```

The message above is a warning indicating that a new file system has been detected and `ipfs` is not in repair mode.

For more information, see “Replay and Repair” on page 85.

Starting up and mounting `ipfs`

At mount time, when `ipfs` detects that the file system was busy (had operations that were not acknowledged by the standby), it reports the following:

```
Oct 09 18:38:30 port /ipfs: startup down summary -
  previously BUSY, setting DIRTY
```

This message is the result of a system being powered off or crashing while having outstanding operations, and indicates a repair will be performed.

Operational Messages

Messages are sent to the console during normal upDisk operations. The role, link, and state of any given mount point are displayed with each output message.

In addition, informational comments may appear to the right of the role, link, and state fields, separated by a hyphen. These informational comments are explained in the rest of this section.

Console messages appear in the following format:

```
servername: mountpoint: role link state - status msgs
```

Role: active, standby, or startup

The `role` field indicates whether this file system has assumed the active, standby, or startup role.

Message	Meaning
`active`	This system is the active one. Applications have read/write access and changes are sent to the standby system.
`standby`	This system is the standby one. Applications have read only access and changes on the active system are propagated to the standby.
`startup`	A role (`active` or `standby`) has not yet been assigned. The `startup` role should rarely be seen.

Table 3 File system roles

Link: up or down

The `link` field indicates whether the link between the active and standby `ipfs` is `up` or `down`, the only two possible states of the link.

State: normal, repair, replay, or summary

The `state` field indicates the state of the system.

Message	Meaning
`normal`	The system is running normally; the link is up and operations are being sent in real time.
`summary`	The link is down and the system is summarizing operations for later replay.
`replay`	The link has been restored and the system is replaying previously summarized operations.
`repair`	Damage was detected and the system is currently repairing it.

Table 4 System states

Comments

You may encounter comments to the right of your role, link, and state messages, separated by a hyphen, which inform you about changes to the role, link, or state (the specific messages are listed later in this section). Such messages are purely informational in nature and simply intended to keep you apprised of system conditions. If, however, an error is logged, upDisk, in most cases, will solve the problem; if the error requires operator intervention, messages will be sent to the console and logs. In addition, if you run the **`udstat`** command, its output will indicate that operator intervention is required.

Dropping the link

Listed below are the circumstances under which a link is dropped:

- Operator shutdown: upDisk or upBeat has detected that a link has failed and instructed `ipfs` to stop using it; or, as the operator, you have intentionally dropped the link.
- Link error: `ipfs` detected a link error before upDisk or upBeat informed `ipfs` about it.
- Active error: an error has occurred during normal operations on the active system. The

link is down and upDisk will failover to the standby system. Typically, you will need to fix the problem and then perform a repair with the **udrepair** command. See "upDisk Command Reference" on page 127.

- Standby error: an error has occurred during normal operations on the standby system. The link is down and, typically, you will need to fix the problem and then perform a repair with the **udrepair** command.

In addition to the above, be aware that a link can be dropped as a side effect of a termination of the link daemon via the system shutting down, upDisk being stopped, or a direct termination of the daemon.

Active/Standby Errors

The active server will fail only if there is an operational or disk error (in which case the active server will failover to the standby); in addition to those errors, the standby server will fail if it runs out of disk space or encounters disk errors or other operational difficulties. In short, if either system finds that it cannot correctly perform operations, it will drop the link and repair will be required. Therefore, it is important to determine *which system is causing the error* before trying to fix it.

Message	Meaning
`(operator shutdown)`	upBeat or upDisk decided that the link was malfunctioning or should not be in place, at which point it shut down the link. (As the operator, you can also shut down the link manually, thus generating this message as well.)
`(active error)`	There is a problem with the active system. `ipfs` will drop the link to avoid possible damage. upBeat will initiate failover.
`(standby error)`	There is a problem with the standby system. `ipfs` will drop the link to avoid possible damage.
`(link daemon killed)`	The upDisk daemon was terminated unexpectedly. This may have been done by the operator, or it may have been done by the system as part of the normal shutdown sequence. If appropriate, the watchdog daemon will restart upDisk; if there are persistent errors, a shutdown is in progress, or the operator killed the watchdog daemon, upDisk will not restart.

Message	Meaning
(link error)	ipfs attempted to use the link and discovered an error. ipfs then terminates communication with the other system and notifies upBeat and upDisk of this event. ipfs noticed the link had a problem before upBeat did. This is atypical and may indicate the daemon is not present. Use **udstat** for more information.

Table 5 Comments in console state output

Miscellaneous Transmission Errors

It is unlikely, but possible, for there to be a protocol error between the two upDisk servers. An XOP error will appear on the console and in /var/adm/messages to notify you of the problem. Normally upDisk will reset the network link, fix any problems, and continue.

If you encounter XOP errors, do the following:

1. Investigate the health of the standby system because XOP errors often indicate a bad disk or other problem on the standby.

2. Verify the health of the system by running **udstat**, because if an XOP error is present, upDisk will usually drop the link, bring up a new link, and run a repair automatically.

3. Even if the above seems to solve the problem and your system appears healthy, please send the XOP error to Continuous Computing's Technical Support via email at support@ccpu.com along with /var/adm/messages from both the active and standby servers so the error can be investigated.

Troubleshooting: other issues

This section describes some common issues that arise when using upDisk and tells how to respond to these events.

File System Access Denied

If your access to the file system is denied, it may have been designated as standby by upDisk. To find out, use the **udstat /ipfs** command at your # prompt. Note that the standby side is read-only and, therefore, all changes must be made to the active system.

File System Out of Disk Space

If one of your systems runs out of disk space, you must remove files on your *active* system to free space and then perform a repair (even if it is the *standby* disk that has run out of space). Note that if it is the standby system's disk that runs out of space the active system will continue to provide full service, but there will be no replication of files on the standby from the active. If the active system's disk is out of space, you can still read, delete, and overwrite data, but you cannot create files or directories.

If one of your systems runs out of disk space, a message similar to those below will appear on your console:

```
Sep 28 17:16:37 port ipfs: [ID 941318 kern.notice] /i:
   The standby is out of space and must be fixed.

Sep 30 03:16:03 left unix: WARNING: /ipfs: File system
   full

Sep 27 10:59:45 left unix: NOTICE: alloc: /ipfs: file
   system full
```

Again, if you get one of the messages above, free space on your active system and perform a repair.

Under some circumstances which depend on the order in which files were created and deleted, you will not be able to complete the repair. If you are unable to complete the repair:

1. Take your standby system offline.
2. Remove files from your standby system to free space on its disk.
3. Perform the repair again.

ipfs Unmount Unsuccessful

`ipfs` will not unmount under certain circumstances. Therefore, you will not be able to successfully run **`/etc/init.d/updisk stop`** (which prevents you from running **`/etc/init.d/updisk start`**). You must successfully run **`stop`** before being able to run **`start`**.

Table 6 lists the reasons `ipfs` will not unmount and instructions for solving the problem. After you have solved the problem, run **`/etc/init.d/updisk start`** to remount the file systems and restart upDisk.

Reason	Solution
A user is either in `ipfs` or using a file in it.	1. Run **`/etc/fuser -c mountpoint`** to find out who is using `ipfs`. (See the manual page for `fuser` for more information about this command). 2. Ask the user to **`cd`** out of `ipfs`, or kill the processes that **`fuser`** lists.
`ipfs` is shared.	1. Use the **`share`** command to find out which file systems are shared via NFS. 2. Unshare exported file systems.
`ipfs` is busy.	Wait for operations to complete before trying to run **`/etc/init.d/updisk stop`** again.

Table 6 `ipfs` unmount solutions

Conflicting File Modification Times (mtime)

File modification times, under normal operations, may appear up to one second off between the active and standby systems. (The modification times may be off only by tens of milliseconds, but some system utilities will round this number up to the nearest second). This is due to operations of the virtual file system. Because of this, modification times may not appear identical on each system, but they will not differ by more than one second.

However, the order of the modification times should be identical on either system. Therefore, utilities that check, sort, and compare modification times should produce equivalent results.

If you use the **`touch`** command in UNIX to set the modification times, those remain absolute.

Repair

If a repair fails for any reason, upDisk tries to repair again immediately. Each time a repair fails, upDisk waits longer before the repair is attempted again, until a maximum of one attempt every four minutes is reached; once that maximum is achieved, the attempts continue indefinitely or until the repair is able to successfully complete. You can manually initiate the repair by running **`udrepair`** on the active system. If a repair has already been started by upDisk, you will get a message that a repair is already in progress.

Split Brain Conditions

Any HA environment is at risk for a split brain condition. We've provided examples below to help you rectify a split brain condition.

A split brain can occur under the following two conditions:

1. All network connections have been severed.

2. upBeat *thinks* all network connections have been severed; for example, because the time-outs have been set too low in `upsuite.conf`.

If a split brain occurs, it is likely that both sides will assume the active role. When the split is rectified, upBeat or upDisk will notice that both sides are active. If this happens, upDisk shuts down the service and then signals a split brain condition both to the operator and to the peer. Under normal operating circumstances, this can only happen after a double failure has occurred, if you are using only two links—multiple communications failures if you are using more than two links—and then been rectified. After one of the conditions described earlier has been detected, you will see a message similar to the following on your console:

```
Aug  8 08:03:24 yoursys: /ipfs: standby down repair - operator intervention now
    required to fix (split)
```

Confirm that the split brain event still exists by running the **`udstat`** command. The output from this command will let you know whether the condition has been rectified or requires your intervention.

upSuite HA does not take automatic corrective action once recovery from a split brain has been detected. As the operator or application developer, you are the only person who can determine what dataset(s) is most accurate.

Recovering from Split Brain

When an upDisk server pair detects a dual active condition, both sides are flagged as split brain and taken offline. As a result, there is no active service.

If either side is flagged as split brain, upDisk flags the other side as split brain as soon as they come into contact. Therefore, to fix a split-brain condition, you must intervene on both sides. If you fix and remove the split-brain flag from only one side, then bring the two systems into contact, upDisk will detect the reamining split-brain flag on the other side and automatically re-flag the fixed side as split brain.

There are two ways to effectively intervene on both systems to fix a split-brain condition:

- Force one side to be active and let it force the other side to be standby. This is the most common technique.
- Explicitly force one side to be active and the other to be standby.

The rest of this section explores each of these techniques.

Technique 1: Force active status on one system

1. Determine which system has the most current data by determining which went down first or last. Do this by inspecting the relevant files and/or system logs, and by using the following command:
   udstat -hh

2. Force the desired side to become active:
   udactive -Af *dataset*

If the active and standby systems are in contact, when you force one side to become active, the other side is forced to become standby immediately. This is the most common way to recover from a split-brain condition.

If the two systems are not in contact, the side you just forced to become active will remember it was forced for as long as it is not restarted again (for example, by a reboot). If it comes back into contact with the other side, that side is forced to become standby. However, if the system that was forced to become active is restarted, it will forget its forced-active status; when the two sides come back into contact, it will be re-flagged as split brain, and you will have to start fixing the split-brain condition again.

Technique 2: Force status on both sides

Use this technique when the two systems are not in contact with each other.

1. Determine which system has the most current data by determining which went down first or last. Do this by inspecting the relevant files and/or system logs, and by using the following command:
   udstat -hh

2. Force the desired system to become active:
   udactive -Af *dataset*

3. Force the other system to become standby:
   udactive -Sf *dataset*

4. Let the systems come back into contact.

For detailed information about the **udactive** and **udstat** commands, see “upDisk Command Reference” on page 127.

Techniques to Avoid When Recovering from Split Brain

There are two things to avoid when trying to fix a split brain condition. By doing either of the following, you will create a new split brain condition:

- Running **`udactive -A -f`** on both systems while the systems are unable to communicate with each other, or;
- Running **`udactive -A -f`** on server A while server B is down, offline, or otherwise unable to communicate with server A, and then rebooting or restarting upDisk on server A. In this case, server A "forgets" that the **`-f`** was run on it, creating a new split brain when B is again operational.

Failover Problems

If one or more of your systems is down:

1. Determine which system has the most current data by determining which went down first or last. Do this by inspecting the relevant files and/or system logs, and by using the **`udstat -hh`** command.

2. Run **`udactive -f`** ***`dataset`*** on the system with the most current data to force it to become active. A repair will start automatically.

svc_max_msg_size Causes Problems when Adding CCPUdisk Package

When installing the `CCPUdisk` software package, you might see a message like the following:

```
Unable to update system parameters dynamically;
You must reboot before starting updisk.
Installation of <CCPUdisk> was successful.
*** IMPORTANT NOTICE ***
        This machine must now be rebooted in order to ensure
        sane operation.  Execute
               shutdown -y -i6 -g0
        and wait for the "Console Login:" prompt.
```

Note particularly the first line, `Unable to update system parameters dynamically`.

As directed in the messages, reboot the machine. Watch for the following in the boot messages:

```
sorry, variable 'svc_max_msg_size' is not defined in the
'rpcmod' module
```

If you will not be configuring HA NFS, or will be using the UDP protocol with HA NFS, you can safely ignore this situation.

However, if you want to use the TCP protocol in an HA NFS system, this message is of concern. HA NFS may fail immediately after the first failover. The failure will be indicated by the following message, which appears on the new active server's console and in `/var/adm/messages`:

```
NOTICE: KRPC: record fragment from client of size(10216)
exceeds maximum (9000). Fragment header was 0x8000807c.
Disconnecting
```

The client size and maximum size in the actual message might be different. The server is not affected by this. Errors, if any, will be on the NFS client. Solaris clients seem to recover well, but Linux clients hang and must be rebooted.

We recommend that you call Continuous Computing Corporation’s technical support if you encounter this problem.

9 Failover

This chapter explains automatic and managed failover.

In this chapter

upDisk Failover Considerations 101
Normal Failover 101
Managed Failover 102
Boot Sequence 104

upDisk Failover Considerations

The split brain issue is more important for upDisk than for many applications. If a standby upDisk server has different data from its active server, the standby is repaired to match the active. In the worst-case scenario: a disk is replaced; the pair of servers is rebooted; the server with the empty disk is selected as active; and the data on the standby is repaired by being deleted.

For these reasons, a server may become active in only these three ways:

- Normal failover
- Cold-start of a server that was active last time it was started
- Operator intervention

In other words, a server cannot automatically become active upon reboot (i.e., upon restart of upBeat/upDisk) unless it was active when it went down. This prevents an inappropriate file system (e.g., a replaced disk) from becoming active.

One side effect of this constraint is that when upDisk is first configured on a pair of servers, you must designate one of them as active.

Normal Failover

In normal operations, the assumption is that there are no double component failures and that failed components are quickly repaired (low MTTR).

If the standby upDisk server fails, there is no disruption in service. The active server summarizes all changes to data in order to replay them to the failed standby server once it is functional again. When it is, the standby is resynchronized with the active server either via replay or

repair. (See “Replay and Repair” on page 85.)

If the active system fails, it quickly fails over to the former standby, making the former standby the new active server. The newly active server summarizes all changes to data. When the formerly active is back in service, it assumes the standby role. The new standby is resynchronized with the new active server either via replay or repair.

Note that if you are gracefully (or otherwise) failing over to the standby system and there are large amounts of data waiting to be written to its local file system, there may be a pause to allow the standby system to process these operations before it becomes the active system. The console will inform you that operations are pending, during which time you can use **`udstat`** for more information. The Standby Throttle Option setting in the `ipfstab` file allows you to limit the number of operations that can remain pending.

Double Failures

Double failures occur when the active system attempts to failover to the standby system but cannot because that system is offline or “dirty”. If this happens, upDisk will be out of service for the dataset. After fixing the problem that caused the failure on the active, you must manually run **`udactive`** on the active system to return it to service.

In the case of a disk failure, SCSIbeat initiates a failover to the standby. If that system is offline or “dirty” and if after the attempted failover the disk becomes operational again, upDisk on the active system will not automatically resume its active role; it will instead fail because its peer is offline, even though the disk problem that originally initiated the failover has been fixed. As a result, both systems are out of service and you must manually run **`udactive`** on the active system to return it to service.

File Locking

For NFS clients only, `ipfs` does not replicate locks. In the case of a failover, you may encounter errors as the files unlock.

Managed Failover

If your configuration changes, you must restart upBeat and everything that interacts with upBeat. The simplest way to deal with configuration changes is to perform a managed failover. This example assumes `left` is the standby system and `right` is the active system at the beginning of the procedure.

The recommended way to instigate a manual failover is to run this command on the active:

udactive -S

or

udactive -Sf

It is not recommended to instigate a manual failover by running **udactive -Af** on the standby. Doing so will possibly lead to operational problems.

Step	Description	System
1.	On the standby system, stop upBeat and everything that interacts with it.	left (currently standby)
	left# **/etc/init.d/updisk stop** left# **/etc/init.d/upbeat stop**	
2.	Edit the configuration file and/or ipfstab to reflect your changes.	left (currently offline)
	left# **vi /etc/upsuite.conf** left# **vi /etc/ipfstab**	
3.	Restart upBeat and everything that interacts with it.	left (currently standby)
	left# **/etc/init.d/upbeat start** left# **/etc/init.d/updisk start**	
4.	On the active system, stop upBeat and everything that interacts with it. This will cause a failover to occur.	right (currently active)
	right# **/etc/init.d/updisk stop** right# **/etc/init.d/upbeat stop**	

Step	Description	System
5.	Edit the configuration file and/or `ipfstab` to reflect your changes.	`right` (currently offline)
	`right# vi /etc/upsuite.conf` `right# vi /etc/ipfstab`	
6.	Restart upBeat and everything that interacts with it.	`right` (currently standby)
	`right# /etc/init.d/upbeat start` `right# /etc/init.d/updisk start`	
7.	If desired, fail back to the original active.	`left` (currently active)
	`left# udactive -S -a`	

Table 7 Performing a managed failover

Boot Sequence

upDisk performs the following steps for each dataset upon startup.This section assumes each upDisk dataset is using an entire local file system.

1. upDisk checks to see whether the local file system is mounted. If so, upDisk alerts the operator and shuts down. This is to avoid problems with people or applications coming in the “back door” and modifying the local file system directly. The file system should only be controlled and accessed through upDisk/`ipfs`. In particular, the file system should not be mounted via /`etc/vfstab`.

2. upDisk checks to see if the local file system has damage (**`fsck -m`**) and notes this information for future reference. Note: It is recommended that UFS file systems be mounted with the **`logging`** option, in which case it is unlikely there will ever be any file system damage.

3. If there was damage, upDisk attempts a repair (**`fsck -o p`**). If, after the repair, there is still damage, upDisk alerts the operator and shuts down.

4. upDisk mounts the local file system.

5. upDisk mounts `ipfs`.

6. upDisk starts its daemon for this dataset.

7. If there was damage, upDisk initiates a repair.

8. The upDisk daemon determines the last state of the file system (active, standby, replay, repair, etc.) If active, the daemon registers with upBeat and requests active status; otherwise, it requests standby status.

upBeat will assign a role asynchronously. If that role matches the request, or if that role is standby, upDisk confirms that role with upBeat and assumes the role. If upDisk requested standby, and upBeat assigned active, upDisk declines by confirming standby.

A standby upDisk server daemon will accept an assignment to failover from standby to active if it is in sync with an active upDisk server daemon. In other words, if the current instance of the standby daemon has ever successfully established a connection with an active upDisk server, and as long as its state is normal (i.e., is not replay, repair, or summary), the daemon will accept an assignment to failover from standby to active.

10 High Availability NFS (HA NFS)

This chapter explains how to use upSuite HA to build a high availability NFS server from a pair of servers.

In this chapter

How Can upDisk Aid My NFS Server? 107
File Handle Persistence During Failover 109
Building an HA NFS Server 110
NFS over UDP vs. TCP 114
Sharing: Subdirectories vs. File Systems 115

How Can upDisk Aid My NFS Server?

Yet another capability of the `ipfs` layer in upDisk is demonstrated in the case of a redundant NFS server running upDisk. The architecture for such a system is shown in Figure 4, where a machine running as an NFS client is connected via a network link(s) to a set of redundant processors and disks, one of which is an active NFS server (in “active” operation) and both of which are running upDisk and upBeat.

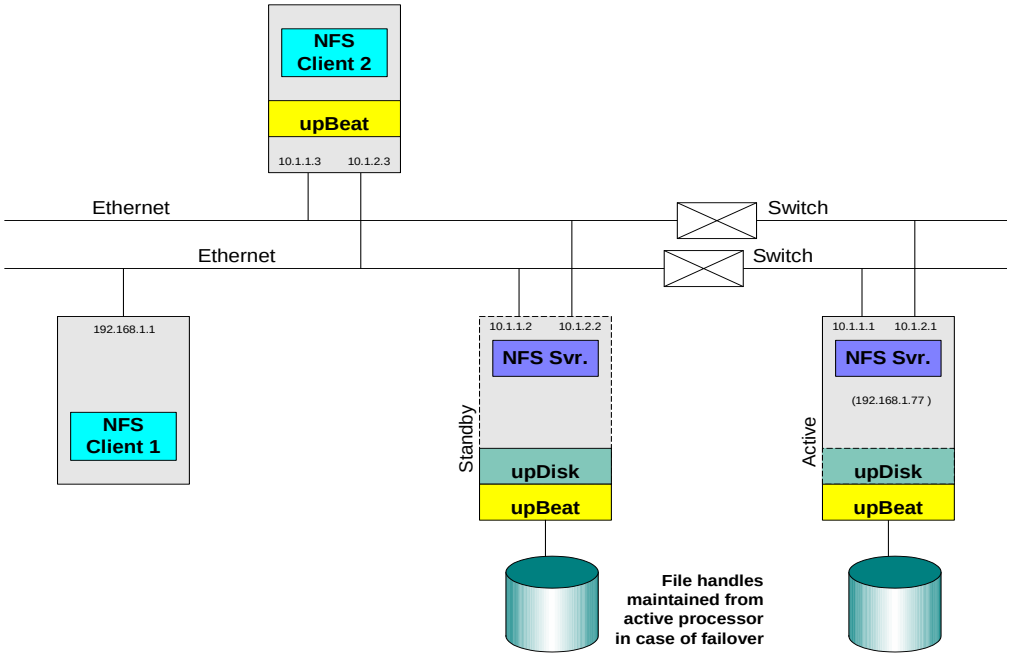

Figure 4 Redundant NFS server running upDisk

Two HA NFS Architectures

There are essentially two ways to construct an HA NFS environment, both of which are illustrated in Figure 4. The architecture you construct will depend on your particular needs.

Note that Client 2 is connected to the system via each of the two Ethernet links and Client 1 is connected to the system via one Ethernet link. In the Client 1 architecture, we assume that the pair of servers are running upDisk as well as upBeat. In the Client 2 architecture, we assume that in addition to the pair of servers running upDisk and upBeat, the client is running upBeat as well.

The Client 1 architecture has an inherent single point of failure, its one connection to the network.

The Client 2 architecture has no single point of failure. In addition, the NFS address may be on the local subnet or any other network. upBeat modifies the routing tables on the client; the client is unaware of NFS address migration during failover (due to upDisk and ARP). (However, be aware that ARP will still notify any non-upBeat clients of NFS address migration during failover.)

File Handle Persistence During Failover

As illustrated in Figure 5, without upSuite HA, the active server associates the file handle 1234 with the `foo.bar` file the NFS client is using. When the active server fails over to the standby system, it associates the file handle 5678 with the `foo.bar` file.

With upSuite HA, `foo.bar` is associated with the file handle 1234. Upon failover to the standby system, the file continues to be associated with the same file handle. Therefore, when the standby server awakens upon the active's failure, the client sees absolutely no difference between it and the active server that was previously providing service.

Without file handle persistence, the failover process would involve closing all files (which likely means shutting down applications), unmounting the file systems (which may require rebooting), remounting the file systems, and reopening all files. Depending on the complexity of the situation, this process could take anywhere from thirty seconds to five minutes and can require a great deal of effort. With upSuite HA NFS, this procedure is avoided.

Note: Refer to "Normal Failover" on page 101for information regarding file locking for NFS clients.

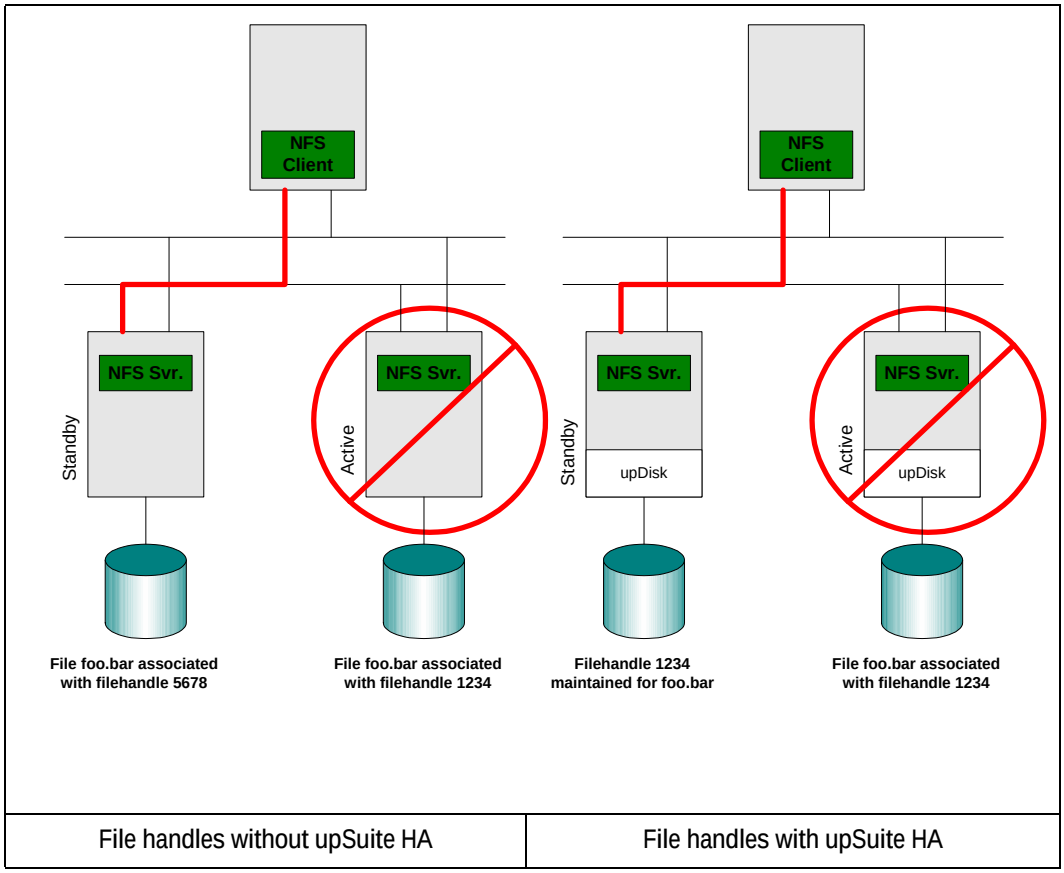

Figure 5 Stale vs. persistent file handles

Building an HA NFS Server

Outlined below are the basic steps necessary to build a high availability NFS server. The remainder of this chapter provides more detailed instructions for how to complete each of the following steps:

1. Add an IP address, interface, and <HANFS> tag to `upsuite.conf` to reflect your HA configuration. The IP address and interface are specified in the <SERVICE_IP> subtag of the <SERVICE> tag. Make sure that `upsuite.conf` contains one, and only one,

<SERVICE_IP> tag per HA NFS service.

CAUTION: If a <SERVICE> tag contains an <HANFS> subtag and does not contain exactly one <SERVICE_IP> subtag, an error will occur and upDisk will shut down.

2. Share the dataset in `/etc/ipfstab`.

3. Optionally, configure clients on redundant networks to run upBeat.

In addition, you will need to decide whether you will run NFS over TCP or UDP, and whether you want to share replicated subdirectories or entire file systems. These issues are discussed in further detail below.

1. Edit the configuration file

Edit `upsuite.conf` to reflect your HA configuration. The IP address you choose will be assumed by the active upDisk server and will be configured or unconfigured as appropriate on the interface you choose.

Note: Each HA NFS upDisk dataset must have its own unique IP address and interface. This "migratory" IP address can be any IP address that is not already in use. In addition, it does not necessarily need to be on the same subnet as the "static" IP addresses in the <NODE> tags. If all of your clients are running upBeat, then the subnet is unimportant. If any of your clients are not running upBeat, you should pick an IP address and an interface (or, more accurately, a sub-interface) that your clients can reach.

Note: Each upDisk dataset must have its own unique PORT.

Perform the following steps on *both* systems, active and standby.

Step	Description	Command
1.	Log in as the superuser.	**`root`**
2.	Go to the configuration file directory.	**`cd /etc/upsuite`**

Step	Description	Command
3.	Edit the <SERVICE> tag of `upsuite.conf` by adding a <SERVICE_IP> subtag (which specifies the IP address and an interface) and an <HANFS> subtag.	**`vi upsuite.conf`**

Typical <SERVICE> *tag:*

```
<SERVICE NAME="updisk:/ipfs" SERVICE_ID="17" TYPE="BASIC" PORT="1776">
        <NODE_REF NODE_ID="1"/>
        <NODE_REF NODE_ID="2"/>
        <LINK NETWORK="Network1"/>
        <LINK NETWORK="Network2"/>
</SERVICE>
```

Same entry edited for HA NFS:

```
<SERVICE NAME="updisk:/ipfs" SERVICE_ID="17" TYPE="BASIC" PORT="1776">
        <NODE_REF NODE_ID="1"/>
        <NODE_REF NODE_ID="2"/>
        <LINK NETWORK="Network1"/>
        <LINK NETWORK="Network2"/>
        <HANFS/>
        <SERVICE_IP IP="192.168.1.77" IF="hme0:17"/>
</SERVICE>
```

CAUTION: You must use one, and only one, <SERVICE_IP> subtag per HA NFS service. Otherwise, an error will occur and upDisk will shut down.

2. Share the dataset in /etc/ipfstab

Edit `/etc/ipfstab` so that the dataset is shared. You can use any of the normal sharing options, e.g., `rw=marketing:engineering,root=engineering`. Note: If your clients have redundant network links, you must specify both of them under share options (separated by a colon). Refer to the following man pages for more detailed information about sharing: `share(1m)` and `share_nfs(1m)`.

Note: Do not use `/etc/dfs/dfstab` to share the dataset! Sharing must happen after upDisk is running; unsharing must happen before upDisk shuts down. You must use `/etc/ipfstab` as described below.

Perform the following steps on *both* systems, active and standby.

Step	Description	Command
1.	Go to the `ipfstab` file directory.	**`cd /etc/upsuite`**
2.	View the end of the file.	**`tail ipfstab`**

	Typical `ipfstab` *entry:*
	
	`# data ipfs ipfs localfs localfs localfs localfs share`
	`# set mount mount mount mount`
	`# name point opts point type device opts opts`
	`#`
	`/ipfs /ipfs - /ipfs ufs c0t0d0s7 logging -`	

3.	If necessary, edit the `ipfstab` file to look similar to the example below. (For complete explanation of each of the fields of the `ipfstab` file, see "ipfstab settings" on page 117.)	**`vi ipfstab`**

	`ipfstab` *edited for sharing:*
	
	`# data ipfs ipfs localfs localfs localfs localfs share`
	`# set mount mount mount mount`
	`# name point opts point type device opts opts`
	`#`
	`/ipfs /ipfs - /ipfs ufs c0t0d0s7 logging rw`	

4.	Start upDisk by either rebooting or manually restarting both the active and standby systems.	Reboot *or* **`/etc/init.d/updisk start`**

3. Configure clients to run upBeat (optional)

If you have clients on redundant networks with the upDisk servers, you can increase the availability of your NFS partitions by running upBeat on the client machines to manage the redundant network connections. To do so, follow these steps:

1. Install `upbeat` on all clients. Refer to the *upSuite Installation Guide* for instructions.

2. Update each server's `upsuite.conf` file to include the client nodes and define heart-

beats between the clients and servers. To do this, add a <NODE> tag for each client, and add <HEARTBEAT> tags to define heartbeats between the client and server machines. You do not need to define heartbeats between the clients. For more information about how to define these tags, see “upSuite Configuration File (upsuite.conf)” on page 145. For an example configuration file, see “Configuration C” on page 170.

3. Copy the `upsuite.conf` file to each client machine. The `upsuite.conf` file should be identical on all servers and clients.

4. Start upBeat on all machines using **`/etc/init.d/upbeat start`**.

5. Start upDisk on all the servers using **`/etc/init.d/updisk start`**.

NFS over UDP vs. TCP

NFS can run over UDP or TCP. Most clients that are TCP-enabled will try that before trying UDP. By default, Solaris NFS servers will serve both UDP and TCP. The default behavior of upDisk HA NFS is to serve UDP; you can modify this by using the PROTOS attribute of the <HANFS> tag in `upsuite.conf`.

NFS over UDP has much better failover characteristics than NFS over TCP; therefore, UDP is recommended. You may designate that the clients use UDP, or you may designate that the servers use UDP, or both. In most cases, it is simpler to configure the servers to use UDP.

On Solaris clients, you can force NFS to use UDP by specifying `proto=udp` as a mount option in `/etc/vfstab`. On Linux clients, you can force NFS to use UDP by specifying `udp` as a mount option in `/etc/fstab`. See the `nfs(5)` man page for details. To find out how to specify UDP on MacOS X or any of the third-party NFS clients for Macintosh or Windows, refer to their documentation.

Running the Solaris nfs daemon with CCPU HA NFS

If you run the upDisk HA NFS and standard Solaris nfs daemons on the same server, the standard Solaris daemons might serve upDisk’s shares, resulting in errors during failover. This issue arises if you configure UDP or TCP but not both. By default, the Solaris NFS daemon (nfsd) serves both TCP and UDP; in comparison, the upDisk HA NFS serves only UDP by default. The Solaris NFS service is typically enabled by placing entries ("shares") in `/etc/dfs/dfstab`.

Solaris clients (and possibly others) try TCP first, then fall back to UDP. If the Solaris nfsd is serving protocols that the updisk HA NFS is not serving (such as TCP) and a client tries to use one of those protocols, the client mounts the upDisk shares from the Solaris nfsd server, which will cause errors during failover.

We recommend you do not run both the HA NFS provided by Continuous Computing and the standard Solaris nfs daemon on the same server; in other words, we recommend against sharing file systems via `/etc/dfs/dfstab`.

The Continuous Computing HA NFS default is UDP only, and the Solaris default is both UDP & TCP. Therefore, if you want to run both servers (not recommended!) you must be careful about which protocols you serve.

One technique would be to change the HA NFS configuration to run both protocols (set the PROTOS attribute to PROTOS="UDP TCP").

Also note that it is critical that you use the HA NFS service IP address or hostname rather than the server's normal hostname in the client-side mount command.

Sharing: Subdirectories vs. File Systems

upDisk can replicate an entire file system or a subdirectory. Usually it is better to replicate an entire file system, but for testing or demonstration purposes, a subdirectory may be appropriate.

Sharing an ancestor of the `ipfs` dataset is not recommended. In other words, if you are replicating a subdirectory and sharing that upDisk dataset, and if you are also sharing any of the upper directories via `/etc/dfs/dfstab`, your results may be unpredictable. This is due to Solaris semantics for sharing a subdirectory of an already-shared directory. This is illustrated in Figure 6.

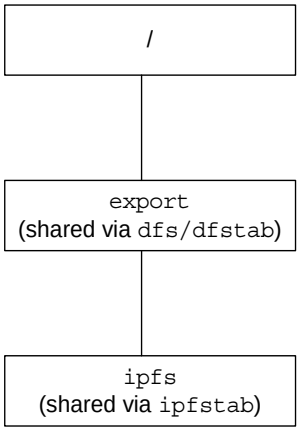

Figure 6 Sharing an ancestor of the `ipfs` dataset (not recommended)

11 The ipfstab File

The `ipfstab` file is a text file that contains a series of one-line entries, each of which consists of a series of settings for a single file system. The `ipfstab` file is used to inform upDisk which `ipfs` file systems to mount. This file is also used by the following commands: **`udstat`**, **`udrepair`**, and **`udactive`**. For explanation of these commands, see "upDisk Command Reference" on page 127.

Normally, upDisk mounts an entire file system. It is also possible to mount only a directory. However, upDisk cannot run `fsck` on a directory; therefore, this practice is recommended for testing purposes only.

This section provides explanations of the `ipfstab` file settings.

Note: The `ipfstab` file resides in the `/etc/upsuite` directory (`/etc/upsuite/ipfstab`); however, for convenience, the symbolic link `/etc/ipfstab` is created during upDisk installation. This symbolic link redirects the short name to the actual file in `/etc/upsuite`. You may use either the `/etc` or the `/etc/upsuite` path to refer to and edit this file. This is similar to how Solaris manages the hosts file in `/etc` and `/etc/inet`.

ipfstab settings

Each line in the `ipfstab` file includes the following settings, in this order:

- data set name
- ipfs mount point
- ipfs mount options
- localfs mount point
- localfs type
- localfs device
- localfs mount options
- share options

The settings are separated by white space; a hyphen (-) indicates a setting for which no value is specified. For example:

```
/mydata /mydata - /mydata ufs cntndnsn rw,noatime,logging -
```

data set name

This uniquely identifies the dataset that is replicated between the upDisk servers. The dataset name must match the service name in the upSuite HA configuration file (`upsuite.conf`); for example, if the dataset name is /`mydata`, then the service name in `upsuite.conf` must be `ipfs:/mydata`. The dataset name must be identical on all servers that replicate this dataset. All other fields may vary from server to server.

ipfs mount point

This field is the mount point for the `ipfs` file system. This is the directory through which users and applications access files.

ipfs mount options

If this field is -, no user-specified mount options are passed to the `ipfs` mount; otherwise, this field is passed as `-o` *`ipfs mount options`*. For information about the available options, see “ipfs Mount Options” on page 121.

localfs mount point

This field is the mount point for the underlying file system. This mount point should be obscured by the `ipfs` mount point, typically by making it the same as the `ipfs` mount point.

The underlying local file system must not be already mounted when upDisk starts. Delete or comment-out entries in `/etc/vfstab` for these local file systems.

We strongly recommend that the local file system mount point must be obscured by the `ipfs` mount point, typically by making both mount points the same, so that `ipfs` hides or obscures the local file system. For debugging or demonstration purposes it may be desirable to have the local file system accessible, but in a production environment it would be disastrous if someone modified the local file system directly while upDisk was running.

localfs type

If you are mounting a directory (for testing purposes), this field should be -.

If you are mounting an entire file system (recommended), this should be the file system type.

Note: upDisk and `ipfs` can use any underlying file system, but the startup script (`/etc/init.d/updisk`) only knows how to repair damage detected by `fsck` on UNIX file systems. If you want to use an underlying file system other than UFS, you will have to modify `/etc/init.d/updisk` as appropriate. (Standard Solaris file systems and standard Disk Suite file systems are both UFS).

localfs device

If `localfs type` is -, this field should be -.

Otherwise, this field is the device to mount on the local file system mount point. Devices are presumed to follow the `{dsk,rdsk}` convention. For example, if you specify `/dev/dsk/c0t0d0s0`, there should be a corresponding `/dev/rdsk/c0t0d0s0` for `fsck` to use.

The device can be a full path or a short cut.

If the device begins with:

/	Indicates the device is a full path.
c	upDisk will look in `/dev/dsk`.
d	upDisk will look in `/dev/md/dsk`.

If the device begins with any other character, or if it was not found in /`dev/dsk` or `/dev/md/dsk`, upDisk will look in `/dev/vx/dsk`. For example:

If you enter:	upDisk will look in:
/dev/dsk/c0t0d0s0	/dev/dsk/c0t0d0s0
c0t0d0s0	/dev/dsk/c0t0d0s0 first, and /dev/vx/dsk/c0t0d0s0 second.
d0	/dev/md/dsk/d0 first, and /dev/vx/dsk/d0 second.
v0	/dev/vx/dsk/v0
disk0/v0	/dev/vx/dsk/disk0/v0

Table 8 Local file system device entries

The letters of the local file system device entry indicate the following:

- **c*n*** SCSI controller number
- **t*n*** SCSI device target number
- **d*n*** Disk number/SCSI LUN (default typically 0)
- **s*n*** Partition slice

For example, if you have one SCSI controller, one disk, and your partition slice is 7, your entry here would be **c0t0d0s7**. Refer to the following Solaris manual pages for more information: `sd (7D), disks (1M), mount (1M), mkfs (1M), newfs (1M)`.

localfs mount options

If `localfs type` is -, this field should be -.

If this field is -, no user-specified mount options are passed to the `localfs mount`; otherwise, this field is passed as `-o` *`localfs mount options`*.

Note: If `localfs type`, `localfs device`, and `localfs mount options` are all -, then `localfs mount point` is treated as a local subdirectory, no `fsck` is done, and no attempt is made to mount a local file system;

`ipfs` just uses the subdirectory.

share options

If this field is -, upDisk does not share the file system.

Otherwise, the entire `ipfs` mount point is shared via NFS with the specified share options, and the NFS daemon and mount daemon are started, if necessary. Exporting subdirectories of the `ipfs` mount point is *not supported* by `ipfstab` and `/etc/init.d/updisk`.

upDisk shares and unshares file systems via NFS depending on the share options. Delete or comment-out entries in `/etc/dfs/dfstab` for either the underlying local file systems or the `ipfs` file systems.

ipfs Mount Options

The mount options of `ipfs` (the filesystem kernel) control various performance and integrity tradeoffs. You set these options by editing the `ipfstab` file.

Options for Returning to the Application

return=[local | sent | recv | remote]

There are four opportunities to return to the application, controlled by the **`return=[local|sent|recv|remote]`** mount option. **`recv`** is the default option.

Table 9 defines each of the return options.

Return option	Action	Notes
local	Return when the local operation completes.	This option provides asynchronous replication. It may be useful when applications have bursty sequences of writes. This option induces a lag in replication and is not appropriate for continuously exceeding network capacity.
sent	Return after `ipfs` sends the operation across the network.	This option provides additional reliability in a network where packets are not lost, (e.g., direct connections), but allows the application to return before it is known to be received by the standby system.
recv	Return when `ipfs` acknowledges the receipt of the operation.	This option is the best tradeoff between performance and reliability because local and remote operations are overlapped and the operation is known to be on the standby. This is called “net synchronous” and is the default behavior.
remote	Return when the remote operation completes.	This option provides the highest reliability (in combination with synchronous operations) at the cost of some performance.

Table 9 Return mount options

Note that the behavior of the local operation and the remote (standby) operation depends on whether synchronous writes are being performed. For example, net synchronous may provide similar reliability as does `fsync()`, but with better performance. For more information about `fsync()` and `O_SYNC`, see the following manual pages: `open(2)`, `fcntl(2)`.

Table 10 illustrates how to configure return behavior.

Changing the ipfstab file's mount options

Typical `ipfstab` *entry*

```
# data   ipfs   ipfs         localfs    localfs  localfs    localfs  share
# set    mount  mount        mount      mount
# name   point  opts         point      type     device     opts     opts
/ipfs    /ipfs  -            /ipfs/     ufs      c0t0d0s7   logging  -
```

Same entry changed to document the default return behaviors

```
/ipfs   /ipfs  return=recv,maxops=1000000,maxmem=10m,throttle=75:100 /ipfs       ufs
   c0t0d0s7  logging  -
```

Table 10 Changing the mount options in the `ipfstab` file

Maximum Operations Option

maxops=#

You can configure the maximum number of replay operations `ipfs` will accrue before flushing operations and switching to repair mode. The default is roughly one million operations (1024*1024) or 10MB of memory, depending on which occurs first.

Memory Allocation Option

maxmem=#.#[k|m|g]

Sets the maximum amount of memory to be allocated for accumulated operations.

Table 11 defines the maximum memory options. Note that units for memory may be specified in decimal values, e.g., 1.5g.

Memory units	Action
k	Sets the specified number in kilobytes.
m	Sets the specified number in megabytes.
g	Sets the specified number in Gigabytes.

Table 11 Maximum memory allocation units

upDisk uses an efficient encoding mechanism for summarizing operations while the link is down. An operation ("op") is approximately 100 bytes. Creates, deletes and other metadata operations allocate an op and memory to store the filename. The memory allocated is proportional to the number of operations and the names involved.

Attribute operations (`chmod`, `chown`, `truncate`) coalesce. The memory allocated is proportional to the number of files on which such operations have been performed. (`open(2)` with the `O_TRUNC` flag or `creat(2)` system calls will generate a metadata and attribute op).

Write operations coalesce whenever possible into an offset and length. The general case of sequential writes into a file will require one op. Random writes generally coalesce over time into one op. The amount of memory allocated is therefore generally proportional to the number of files to which writes have been performed.

Examples are provided below.

- Creating 10,000 files with names "0000" through "9999" would allocate 10,000 x 100 bytes for ops, 5 x 10,000 for names, equaling a total of 1,050,000 bytes.
- Deleting 10,000 files with names "0000" through "9999" would allocate the same amount of memory.
- Writing 1 byte to 10,000 different files would allocate 1,000,000 bytes for ops.
- Writing 1MB to 10,000 files would allocate 1,000,000 bytes for ops.
- Writing 1TB to 10 files would allocate 1,000 bytes for ops.

Standby Throttle Option

throttle=*low*:*high*

`ipfs` has a multi-threaded standby implementation which allows it to acknowledge operations from the active before they are performed (e.g., **`return=local/sent/recv`**). In situations where the active system presents many operations faster than the standby can perform them, the drain time for these operations on the standby may be significant during failover. In other words, it may take several seconds to drain thousands of operations.

To limit the failover drain time a standby "throttle" (implemented with a "high water/low water" mechanism) enables you to configure the number of outstanding operations on the standby. The defaults are 75 for the low water mark and 100 for the high water mark (e.g., **`throttle=75:100`**).

Set this option using **`throttle=`*`low`*`:`*`high`***. The values must be set in integer decimal values with high being greater than low. Larger values may increase performance by allowing more operations to be queued on the standby, the tradeoff being a longer drain time during

failover. To disable the throttle, use **`throttle=0:0`**.

Synchronous/Asynchronous Option

+sync - operations are forced synchronous

-sync - operations are forced non-synchronous

You can use `+sync` and `-sync` to override application control of synchronous operations to the underlying file system. The absence of either of these options leaves the application in control. In particular, NFS servers will still be synchronous.

`+sync` forces all directory and IO operations to be synchronous.

`-sync` forces all directory and IO operations to be cached (non-synchronous) and can be used for database and NFS server applications to greatly increase performance. Since there is an up-to-date replicated copy of the data on another system, no loss of data will occur in the event of a system or component failure.

Standby Modification Time Option

mtime

Controls the setting of modification time on the standby. The default behavior of upDisk is not to use the `mtime` option. Without this option, files on the standby may have a later modification time than those on the active, due to replication delay. However, each file on the standby will have the correct modification time relative to other files on the standby.

With this option turned on, the modification time on the standby is made exactly the same as on the active. This requires upDisk to perform an extra attribute operation, which can affect performance.

12 upDisk Command Reference

This chapter explains upDisk's commands. The commands are organized alphabetically and include usage, a description, options, relevant files, and other related commands. You must be logged in as superuser (**`root`**) to use these commands.

In all commands, ***`dataset`*** and ***`mountpoint`*** are as you have defined them in the `ipfstab` file.

In this chapter

udactive 128
udrepair 131
udstat 132
updisk (admin command) 137
updisk (startup script) 139

udactive

NAME

udactive

USAGE

/usr/sbin/udactive [-A | -S] [-f] {*dataset* | -a | -d *dataset* | -m *mountpoint*}

DESCRIPTION

This command is used to request or force changes in active/standby status.

OPTIONS

The **-A** flag makes the system on which the command is run the active system. This is the default behavior. This command causes upDisk go active as long as upBeat is not reporting another active for that dataset on the network and as long as there are no operator issues (e.g., split brain). upDisk registers (or reregisters) with upBeat to become active and waits to be informed it is active. The primary reason for this flag is to indicate which side should be the active after you have first installed upBeat.

The **-S** flag makes the system on which the command is run the standby system.

The **-f** flag forces the system on which the command is run to become either active or standby, depending on the required argument you use. Including this flag forces upDisk to clear any problems (e.g., split brain) and instructs upDisk to tell the other side (if it is available) to become the status opposite of that of the current machine; for example, if the current machine is active, this command instructs the peer to go standby and vice versa. Then the situation is identical to the **-A** flag: upDisk registers (or reregisters) with upBeat to become active and waits for instructions. Use this command with caution. Note that this command will succeed even if the standby is not in sync with the active. This flag is primarily for recovering from split brains or other problems that require operator intervention.

Without **-f**, this command may refuse to do what is asked based on the current system status from upBeat.

The **<*dataset*>** argument limits the operation to the specified dataset.

The **-a** flag specifies *all* datasets in `ipfstab`.

The **-d** flag is the default. The following two commands yield the same information: **udactive *dataset*** and **udactive -d *dataset***

The **-m** flag allows you to specify the mount point of the dataset. This is only useful if the two are different from one another.

FILES

`/etc/ipfstab`,
`/etc/upsuite/upsuite.conf`

SEE ALSO

`/usr/sbin/udstat`, `/usr/sbin/udrepair`

"Normal Failover" on page 101

udrepair

NAME

udrepair

USAGE

/usr/sbin/udrepair {*dataset* | -a | -d *dataset* | -m *mountpoint*}

DESCRIPTION

This command initiates a repair *as long as* `updisk` *is running*.

OPTIONS

The ***dataset*** argument limits the status check to the specified dataset only.

Refer to the Options of the **udactive** command for explanation of the **-a**, **-d**, and **-m** flags as their usage is identical with **udrepair**.

FILES

`/etc/ipfstab`

SEE ALSO

`/usr/sbin/udstat, /usr/sbin/udactive, /etc/upsuite/upsuite.conf`

udstat

NAME

udstat

USAGE

/usr/sbin/udstat [-i] [-h[h[h]]]
{*dataset* | -a | -d *dataset* |
-m *mountpoint*}

DESCRIPTION

This command provides information about the dataset.

Note: Using **udstat** without any arguments is the same as **udstat -i**.

OPTIONS

The ***dataset*** argument limits the status check to the specified dataset only.

The **-i** flag provides information about the dataset.

Refer to the Options of the **udactive** command for explanation of the **-a**, **-d**, and **-m** flags as their usage is identical.

Note the **-a** option will print the dataset and mount point as follows:

```
<dataset> <mountpoint>: ACTIVE normal UP clean
0 0
```

Note: You will see both a dataset and mount point listed together only if they are *not* named identically. Otherwise, you will see only the dataset name.

OPTIONS (cont.)

Below are the possible messages you will see for each column of output.

`ACTIVE/STANDBY/STARTUP`

Indicates the role of the dataset.

`ACTIVE` indicates this system is the active for the specified dataset.

`STANDBY` indicates the system is the standby for the specified dataset.

`STARTUP` indicates the system is starting up. Under normal operations, this is a transitory state. If a `STARTUP` message persists, it usually means the daemons are not running; this will be accompanied by the `nodaemon` output described later in this section.

`normal/summary/replay/repair`

Indicates the state of the system.

`normal` indicates the system is running normally.

`repair` indicates that a repair is currently taking place.

`replay` indicates that the active system is currently sending material to update the recently failed standby system.

`summary` indicates that the active is not currently replicating to the standby. The active is creating a log of all operations that will have to be performed on the standby when it is available again. This message typically indicates the standby is offline.

OPTIONS (cont.)

`UP/DOWN`

Indicates whether the link between the active and the standby `ipfs` is up or down.

`clean/dirty/busy`

`clean` indicates that each system is in sync with the other.

`dirty` indicates that one side or the other (or both) believe a repair is necessary.

`busy` indicates the system is working but may be behind the other by a few transactions.

`0 0`

These numbers indicate the number of deferred operations and the number of replay operations, respectively. If there are deferred or replay operations outstanding, both of these numbers will be displayed. (If there are no outstanding operations, no values will be displayed). Deferred operations accumulate if the standby is offline or if the server is busy completing a repair. Deferred operations are moved to the replay list when the standby comes back online or as a result of repair activity. During a repair, it is possible to see both replay and deferred operations because the server may briefly defer operations for parts of the file system.

`stopped`

`stopped` indicates that upDisk has stopped operations.

`operator/splitbrain`

`operator/splitbrain` indicates that operator intervention is required and that a splitbrain condition is the reason.

`nodaemon`

`nodaemon` indicates that upDisk is not running for this dataset.

OPTIONS cont'd

`dataset not mounted`

`dataset not mounted` indicates that the dataset is listed in `ipfstab` but is not currently available.

`offline`

The local upDisk daemon is running, this is not the active server for this dataset, and the local daemon is not in contact with the peer daemon. There are several reasons `offline` might be displayed:

- upBeat has detected a problem with a disk that this service depends on (as configured in `upsuite.conf`) and has told this service to go offline.
- If a repair fails, upDisk repeatedly waits and retries the repair. During the waiting periods between failed repair attempts, the standby is offline.
- In a split-brain situation, both servers go active; when the split brain is repaired, upDisk notices the dual active condition and takes both sides offline.

To find out why `offline` is displayed, consult the log files.

Examples:

```
/ipfs: STANDBY summary DOWN dirty
offline
```

```
/ipfs: STANDBY summary DOWN clean
offline
```

```
/ipfs: STANDBY repair DOWN dirty
offline
```

```
/ipfs: STARTUP repair DOWN dirty
operator/splitbrain offline
```

The **`-h`** flag returns history information. The history entries are updated every time there is a change in the state among startup, active, and standby. If a given component is without a role and becomes standby, the history is changed, and vice versa. Otherwise, the current entry is updated.

OPTIONS cont'd

The **-h** flag returns the most current change. Output will look similar to the following:

```
left# udstat -h /ipfs
2001/10/01 14:25:10      4710.45 00000448bcda1494 00000000
  ACTIVE (replay)
  +replay -repair +busy -dirty -fsck -odirty -new -onew
  gday (+session +epoch) def/rep 0 501
```

The **-hh** flag returns a predefined number (currently eight) of recent changes made.

The **-hhh** flag returns the same information as **-hh** and includes the times the changes were made.

FILES

`/etc/ipfstab`

SEE ALSO

`/usr/sbin/udrepair, /usr/sbin/udactive`

updisk (admin command)

NAME

updisk (admin command)

USAGE

updisk [[-c] [-d] [-F] [-s | -S] [-v] *dataset mountpoint*] [-h] [-V]

DESCRIPTION

This command starts the watchdog process and the upDisk daemon; these are the core upDisk executables.

Typically this command is run by the upDisk startup script and not manually.

OPTIONS

Running this command without any arguments causes upDisk to run as a daemon. Program messages are sent to `syslog`.

Including the **`-c`** flag causes program messages to be sent to your console in addition to any other place they are being sent, for example, the `syslog` if upDisk is running as a daemon.

Including the **`-d`** flag runs upDisk in debug mode, so that upDisk runs in the foreground. Messages that would usually go to `syslog` will output to the screen along with additional debugging information.

The **`-F`**(sck) flag causes a repair.

The **`-s`** flag forces program messages to be sent to `syslog` even if, for example, you have already run the command with the **`-d`** flag which routes messages to the console rather than `syslog`.

The **`-S`** flag causes program messages to not be sent to `syslog` even if you have already run a command which routes them there, for example, if upDisk is running as a daemon.

The **`-v`**(erbose) flag sends more program messages with more detail.

The ***`dataset`*** and ***`mountpoint`*** arguments specify the dataset and mount point from the `ipfstab` file.

The **`-h`**(elp) flag prints the usage of this command.

The **`-V`**(ersion) flag prints the version of upDisk.

FILES

`/etc/ipfstab, /etc/upsuite.conf`

SEE ALSO

`/usr/sbin/upbeat, /usr/sbin/udrepair, /usr/sbin/udstat, /usr/sbin/udactive`

updisk (startup script)

NAME

`updisk` (startup script)

USAGE

`/etc/init.d/updisk {start | stop}`

DESCRIPTION

This command is run automatically upon Solaris reboot and invokes **`/usr/sbin/updisk`**. If you have installed upDisk and want to begin using it without rebooting, then use the above command with the **`start`** argument.

OPTIONS

`start` starts upDisk for all datasets in `ipfstab` and mounts the `ipfs` and underlying file systems.

`stop` stops upDisk for all datasets in `ipfstab` and unmounts the `ipfs` and underlying file systems.

FILES

`/etc/ipfstab`

SEE ALSO

`/usr/sbin/updisk`

`/etc/rc2.d/K98updisk`

`/etc/rc3.d/S98updisk`

`init(1M)`, `inittab(4)`

13 upDisk API

This chapter explains upDisk's optional application programming interface (API).

In this chapter

API Overview 141
Function Call Guidelines 141
Sample API Code 141

API Overview

Using upDisk's API is optional. Most applications can simply use the upDisk file system without modifications as they would any other file system. However, upDisk's API consists of one `ioctl` function call which allows you to learn the following information:

- Whether you are on a valid upDisk file system.
- Whether you are on the active file system.

Function Call Guidelines

All programs must include `ipfs.h`. This file is stored in the directory `/opt/upsuite/include`.

Sample API Code

Below is a sample program using upDisk's API function call.

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/param.h>

#include <ipfs.h>
```

```
char *fs_state2string(int fs_state);
char *fs_role2string(int fs_state);

int
main(int argc, char *argv[])
{
    char    *path, handle[MAXPATHLEN + 1];
    int     n, fd, status;
    int     wait;

    /*
     * This program determines whether a mount point is an IPFS (upDisk)
     * mount point, or whether a file or directory is in an IPFS
     * file system.
     */
    if (argc != 2) {
        fprintf(stderr, “Usage: %s file | directory | mount_point\n”, argv[0]);
        exit (1);
    }

    /*
     * Every IPFS file system has an entry at the top level
     * which is designated by IPFS_HANDLE and which can be opened
     * to determine status.
     */
    n = snprintf(handle, MAXPATHLEN, “%s/%s”, argv[1], IPFS_HANDLE);
    if ((n > MAXPATHLEN) || (n < 0)) {
        fprintf(stderr, “path too long or snprintf() failed\n”);
        exit (1);
    }

    /*
     * Try the handle first in case we were handed a mount
     * point; then try the path directly in case we were handed
     * a file or directory.
     */
    path = handle;
    if ((fd = open(path, O_RDONLY)) == -1) {
        if ((errno == ENOENT) || (errno == ENOTDIR)) {
            path = argv[1];
            if ((fd = open(path, O_RDONLY)) == -1) {
                fprintf(stderr, “open(%s) failed: errno %d %s\n”,
                  path, errno, strerror(errno));
                exit (1);
            }
```

```
        } else {
            fprintf(stderr, “open(%s) failed: errno %d %s\n”,
              path, errno, strerror(errno));
            exit (1);
        }
    }

    /*
     * Try to get IPFS status. ENOTTY tells us this is not an IPFS
     * file system; any other error is reported as an error.
     *
     * If wait is 0, return status immediately.
     */
    wait = 0;
    if ((status = ioctl(fd, IPFS_STATUS, wait)) == -1) {
        if (errno == ENOTTY) {
            printf(“%s: not an IPFS file system\n”, path);
        } else {
            fprintf(stderr, “ioctl(IPFS_STATUS) failed: errno %d %s\n”,
              errno, strerror(errno));
            exit (1);
        }
    } else {
        printf(“%s: IPFS file system:\n”, path);
        printf(“\t%s\n”, fs_state2string(IPFS_S_STATE(status)));
        printf(“\t%s\n”, fs_role2string(IPFS_S_ROLE(status)));
        printf(“\t%s\n”, status & IPFS_S_ACTIVE ? “active” : “not active”);
        printf(“\t%s\n”, status & IPFS_S_LINK ? “link up” : “link down”);
        printf(“\t%s\n”, status & IPFS_S_SYNC ? “clean” : “dirty”);
        printf(“\t%s\n”, status & IPFS_S_STOP ? “stopped” : “not stopped”);
    }

    (void)close(fd);

    exit (0);
}

char *
fs_state2string(int fs_state)
{
    char    *state_string;

    switch (fs_state) {
```

```
        case S_NORMAL:
            state_string = “normal”;
            break;
        case S_SUMMARY:
            state_string = “summary”;
            break;
        case S_REPLAY:
            state_string = “replay”;
            break;
        case S_REPAIR:
            state_string = “repair”;
            break;
        default:
            state_string = “unknown state”;
            break;
        }

        return (state_string);
    }

    char *
    fs_role2string(int fs_role)
    {
        char    *role_string;

        switch (fs_role) {
        case S_STARTUP:
            role_string = “startup”;
            break;
        case S_STANDBY:
            role_string = “standby”;
            break;
        case S_ACTIVE:
            role_string = “active”;
            break;
        default:
            role_string = “unknown role”;
            break;
        }

        return (role_string);
    }
```

14 upSuite Configuration File (upsuite.conf)

This chapter describes the upSuite HA configuration file, `upsuite.conf`. The configuration file, which is in XML format, contains settings that govern various aspects of the behavior of upSuite. You must edit `upsuite.conf` to reflect the settings of your system. To help you through this process, this chapter includes descriptions of each setting in the configuration file. In addition, three sample configurations are included to be used as guides for creating your own upSuite architectures.

In this chapter

The Configuration File 145
Editing upsuite.conf 147
<UpSuiteConfig> tag 148
<UPBEAT> tag 148
<NETWORK> tag 149
<NODE> tag 149
<HEARTBEAT> tag 152
<SERVICE> tag 154
Configuration File Example 159
Sample Configurations 161

The Configuration File

The upSuite HA configuration file, `upsuite.conf`, is an XML file that contains settings you can modify to customize the behavior of upSuite software. Several example configuration files are provided with the upSuite software in the `/etc/upsuite/examples` directory:

- `upbeat.xml` contains example settings relevant to upBeat.
- `updisk.xml` contains example settings relevant to upDisk.
- `upsuite.xml` contains example settings for all possible upSuite configuration options.

You can copy one of these files to `/etc/upsuite/upsuite.conf` and use it as the basis for a new configuration file.

ubManager has its own configuration file. The same directory contains example files for ubManager:

- `ubmgr.xml`
- `sample.4ubmgr.xml`

As an XML file, `upsuite.conf` contains tags and attributes. Each tag or attribute corresponds to an item you can configure. The file's root tag, <UpSuiteConfig>, has several subtags that correspond to various configuration items such as networks, nodes, and services. The rest of this chapter describes these tags in detail.

The `upsuite.conf` file resides in the `/etc/upsuite` directory (`/etc/upsuite/upsuite.conf`); however, for convenience, the symbolic link `/etc/upsuite.conf` is created during installation. This symbolic link redirects the short name to the actual file in `/etc/upsuite`. You may use either the `/etc` or the `/etc/upsuite` path to refer to and edit this file. This is similar to how Solaris manages the hosts file in `/etc` and `/etc/inet`.

Managing the Configuration File on Multiple Machines

You must configure upSuite on all active and standby systems in your high-availability system. The configuration file must be the same on all machines. Each machine uses its IP addresses to determine which node it is.

If you change your configuration, you must change or copy the configuration file to every other system, and you must restart upBeat and everything that interacts with upBeat.

Using XML in the Configuration File

XML follows strict guidelines. For example, all characters are case-sensitive. There are two types of attributes: required and optional. The required attributes must be present for the configuration file to function; the optional attributes can be omitted. Each of the attributes is explained in detail in this chapter.

Avoiding Name Conflicts in the Configuration File

No two nodes or services can have the same name or ID. In other words, the NAME, and NODE_ID or SERVICE_ID attributes of any given <NODE> or <SERVICE> tag must be set to different values from the same attributes in other tags of the same type. For example, a <NODE> tag and a <SERVICE> tag could both contain NAME attributes set to the same value, but no two <SERVICE> tags should contain NAME attributes set to the same value.

Within a node, partition names must be unique. That is, no two <PARTITION> subtags within a given <NODE> tag can have NAME attributes set to the same value. However, if the <PARTITION> subtags are within different <NODE> tags, the value of NAME can be identical if desired.

Example: Valid namespace use

The following partition names, while identical, are not in conflict because they occur in different <NODE> tags:

```
<NODE NAME="node1" NODE_ID="1">
   <PARTITION NAME="data" DEVICE="/dev/rdsk/c0t1d0s2"/>
</NODE>
<NODE NAME="node2" NODE_ID="2">
   <PARTITION NAME="data" DEVICE="/dev/rdsk/c0t1d0s2"/>
</NODE>
```

Example: Invalid namespace use

The following example shows partition names that are invalid, because the two identical names occur within one <NODE> tag:

```
<NODE NAME="node1" NODE_ID="1">
    <PARTITION NAME="data" DEVICE="/dev/rdsk/c0t1d0s2"/>
    <PARTITION NAME="data" DEVICE="/dev/rdsk/c0t2d0s2"/>
</NODE>
```

Editing upsuite.conf

If your configuration changes, you must edit the `upsuite.conf` file and restart upBeat and all of its services. To do so, follow these steps:

1. Save the existing `upsuite.conf` to a backup file. For example:
   `cp upsuite.conf upsuite.sav`

2. Edit `upsuite.conf` to reflect your configuration changes. For example:
   `vi upsuite.conf`

3. Stop all clients of upBeat on the standby system. For example, if ubManager and upDisk are clients of upBeat:
   `/etc/init.d/ubmgr stop`
   `/etc/init.d/updisk stop`

4. Stop all clients of upBeat on the active system. For example, if ubManager and upDisk are clients of upBeat:
   /etc/init.d/ubmgr stop
   /etc/init.d/updisk stop

5. Stop upBeat on the standby system:
   /etc/init.d/upbeat stop

6. Stop upBeat on the active system:
   /etc/init.d/upbeat stop

7. Start upBeat on all systems in your configuration:
   /etc/init.d/upbeat start

8. Start any services of upBeat on all systems in your configuration. For example:
   /etc/init.d/updisk start
   /etc/init.d/ubmgr start

<UpSuiteConfig> tag

The <UpSuiteConfig> tag is the root tag of the configuration file. Everything within this tag is parsed to extract configuration settings; anything outside this tag is ignored.

The <UpSuiteConfig> tag has one required attribute, VERSION, which must be set to the version number of the software you are using; in the current release, this must be set to "2" (not 2.0, 2.0.0, or any variation):

```
<UpSuiteConfig VERSION="2">
...
</UpSuiteConfig>
```

<UPBEAT> tag

The <UPBEAT> tag is optional. This tag has one optional attribute, STARTUPDELAY_SEC, which specifies the startup period in seconds. If the <UPBEAT> tag is not used or the attribute is not set, the default is 5 seconds. The following example shows the <UPBEAT> tag with the attribute set:

```
<UPBEAT STARTUPDELAY_SEC="7"/>
```

During the startup delay period specified in the <UPBEAT> tag, `ubInit()` will block.

<NETWORK> tag

The <NETWORK> tag defines your networks. There is no minimum or maximum number of networks. A typical set of <NETWORK> tags looks similar to the following:

```
<NETWORK NAME="Network1" DESCRIPTION="192.168.1.0/24"/>
<NETWORK NAME="Network2" DESCRIPTION="192.168.2.0/24"/>
```

NAME attribute

Required. Provide a name for the network. The value of the NAME attribute for each network must be different from the NAME attribute in any other <NETWORK> tag.

DESCRIPTION attribute

The DESCRIPTION attribute is optional. Use it to describe your network in any way you choose. In this example, the IP address is used.

<NODE> tag

The <NODE> tag defines your nodes and their interfaces. In addition, the <NODE> tag may specify partitions, which you can use to monitor your SCSI disks. A <NODE> tag with partitions specified would look similar to the following:

```
<NODE NAME="left" NODE_ID="1"
 DESCRIPTION="SPARC CP1500 Solaris 2.7">
   <INTERFACE NAME="hme0" NETWORK="Network1" IP="192.168.1.2"/>
   <INTERFACE NAME="hme1" NETWORK="Network2" IP="192.168.2.2"/>
   <PARTITION NAME="system" DEVICE="/dev/rdsk/c0t0d0s0"
          TIMEOUT_MSEC="2000" FREQ_MSEC="600"/>
   <PARTITION NAME="one" DEVICE="/dev/rdsk/c0t1d0s5"
          TIMEOUT_MSEC="2000" FREQ_MSEC="600"/>
   <PARTITION NAME="two" DEVICE="/dev/rdsk/c0t1d0s4"
          TIMEOUT_MSEC="3000" FREQ_MSEC="700"/>
</NODE>
```

```
<NODE NAME="right" NODE_ID="2"
 DESCRIPTION="SPARC CP1500 Solaris 2.7">
   <INTERFACE NAME="hme0" NETWORK="Network1" IP="192.168.1.3"/>
   <INTERFACE NAME="hme1" NETWORK="Network2" IP="192.168.2.3"/>
   <PARTITION NAME="system" DEVICE="/dev/rdsk/c0t0d0s0"
          TIMEOUT_MSEC="2000" FREQ_MSEC="600"/>
   <PARTITION NAME="one" DEVICE="/dev/rdsk/c0t1d0s4"
          TIMEOUT_MSEC="2000" FREQ_MSEC="600"/>
   <PARTITION NAME="two" DEVICE="/dev/rdsk/c0t1d0s5"
          TIMEOUT_MSEC="3000" FREQ_MSEC="700"/>
</NODE>
```

A given system locates its own node ID number by comparing its IP addresses to those listed for each node in `upsuite.conf`. A system must have all the IP addresses that are listed for a node, and it can have more IP addresses as well. upBeat issues an error in three situations: it cannot find a matching IP address, the current system has some but not all IP addresses listed for a node, or some of the current system's addresses are listed under one node and some are listed under another node.

NAME attribute

Required. This names the node. The name must be unique; that is, the NAME attributes of all <NODE> tags must be different. In a <NODE> tag, the NAME attribute does not necessarily have to match the name in the host file.

NODE_ID attribute

Required. The numerical ID of the node. Choose any number you like, as long as it is unique (among all the <NODE> tags) and a positive 32-bit integer greater than or equal to 1.

DESCRIPTION attribute

Optional. This is your desired description of the given node.

<INTERFACE> subtag of the <NODE> tag

The <INTERFACE> subtag specifies the name, network, and IP address (IPv4 only) of the interface.

NAME attribute

Required. Names the given interface. The name must be unique; that is, the NAME attributes of all <INTERFACE> tags within a given <NODE> tag must be different.

NETWORK attribute

Required. Used by the <LINK> subtag in services and heartbeats to tie together two IP addresses from a pair of servers.

IP attribute

Required, unless the HOST attribute is used. This is a host name or IP address.

HOST attribute

You may replace the IP address attribute with the HOST attribute, in which you would specify the host name of your server; for example, `HOST="serverA"`.

Using HOST may be convenient for testing and evaluation purposes, but we recommend that HA deployments rely on the IP address to reduce the reliance on unresponsive name services. Note that upBeat will reject hosts with multiple IP addresses. Therefore, if replacing IP with the HOST attribute, be sure that the host name specified is not associated with more than one IP address.

While the value of IP is passed to `inet_addr()`, the value of `HOST` is passed to `gethostbyname()`.

<PARTITION> subtag of the <NODE> tag

Specifies the name, device, and timeouts of a partition.

Use the <PARTITION> subtag only if you need to monitor SCSI disks in a Solaris environment; this is due to Solaris' SCSI disk driver, which might not report an unresponsive disk in a timely manner. The wait may be too long for your specific needs. If you need to monitor one or more particular SCSI disks more closely, in addition to adding the <PARTITION> subtag to the <NODE> tag, you must add a <SCSIBEAT> subtag under the <SERVICE> tag (as described in "<SERVICE> tag" on page 154). SCSIbeat initiates a failover if a SCSI disk becomes unresponsive.

The Solaris interface requires that you specify a partition, even though you may only want to monitor a disk. Therefore, to monitor the disk `c0t0d0`, you must watch one of its partitions (`c0t0d0s0` through `c0t0d0s7`). If your application depends on two partitions on the same disk, for example `c0t0d0s0` as the system partition and `c0t0d0s5` as the data partition, you only need to monitor one of the partitions.

NAME attribute

Required. Gives a name to the partition. The name must be unique within the node; that is, the NAME attributes of all <PARTITION> subtags of a given <NODE> tag must be set to different values. The systems in your HA configuration do not all have to use the same physical partition, as illustrated by the example earlier in this section.

DEVICE attribute

Required. This is the device name of the given partition. It must be unique.

TIMEOUT_MSEC attribute

Optional. This is the amount of time SCSIbeat will wait to hear from the disk before issuing a disk failure. The default is 2000 ms.

FREQ_MSEC attribute

Optional. This is the frequency with which a SCSIbeat inquiry is sent to the disk. The default is 950 ms. Note that because SCSIbeat causes disk activity, the disk LEDs may flicker or remain lit depending on the disk you are using. For more information about SCSIbeat, see “SCSIbeat” on page 11.

<HEARTBEAT> tag

A <HEARTBEAT> tag instructs upBeat to send heartbeats between two systems (nodes) over specified links. Each system uses these heartbeats to determine the health of the network connections. When all links to another node are down, the node is declared down. If a server with an active service is declared down, the standby will be told to become active for that service.

A <HEARTBEAT> tag with sockets set to route would look similar to the following:

```
<HEARTBEAT
    NAME="left -- right"
    TYPE="POINT_TO_POINT"
    TIMEOUT_MSEC="500"
    RESEND_MSEC="150">
    <NODE_REF NODE_ID="1"/>
    <NODE_REF NODE_ID="2"/>
    <LINK NETWORK="Network1" ROUTE=”ROUTE”/>
    <LINK NETWORK="Network2" ROUTE=”ROUTE”/>
</HEARTBEAT>
```

NAME attribute

Required. This names the heartbeat. The name must be unique; that is, the NAME attributes of all <HEARTBEAT> tags must be different.

TYPE attribute

Required. Must be POINT_TO_POINT. Future releases of upBeat will support broadcast and multicast heartbeats.

TIMEOUT_MSEC attribute

Required. Controls the time it takes the server to determine if a link to its peer is down. The value is specified in milliseconds. As latency increases, you should increase the time. We recommend starting with between a 3:1 or 4:1 ratio of TIMEOUT_MSEC to RESEND_MSEC. If you are using a MAN or WAN, you will need to experiment with your ratios for your particular needs.

RESEND_MSEC attribute

Required. The time between sending out heartbeat packets. The value is specified in milliseconds. As latency increases, you should increase the time. We recommend starting with between a 3:1 or 4:1 ratio of TIMEOUT_MSEC to RESEND_MSEC. If you are using a MAN or WAN, you will need to experiment with your ratios for your particular needs.

<NODE_REF> subtag of <HEARTBEAT>

The <NODE_REF> subtag of the <HEARTBEAT> tag refers to the nodes as you have defined them in the NODE_ID attribute of the <NODE> tag. Each <HEARTBEAT> tag must contain two <NODE_REF> subtags.

NODE_ID attribute

Required. This is the node as it is defined in the NODE_ID attribute of the <NODE> tag.

<LINK> subtag of <HEARTBEAT>

Required. Defines the links over which heartbeats are to be sent. The <HEARTBEAT> tag must contain at least one <LINK> subtag, although two or more are recommended. A high-availability system requires at least two networks.

NETWORK attribute

Required. Name of the network. This allows the heartbeats to associate IP addresses from two different nodes. This is a reference to the NETWORK attribute of the <INTERFACE> subtag of the <NODE> tag.

ROUTE attribute

Optional. Specifying ROUTE enables sockets to route. This enables you to establish a heartbeat between nodes that are on different subnets or that are separated by routers. Packets will be routed according to your node's routing tables. You must ensure there are separate network paths for each link and that packets for one link cannot be routed over a path for another link (refer to "<SERVICE> tag" on page 154 for information about configuring links). Incorrect routing tables may create failure detection problems or a single point of failure, leaving your system vulnerable to a split brain condition.

If you do not use the ROUTE attribute, the default behavior is to not route (specified via the

SO_DONTROUTE socket option), meaning that packets can only be sent to other nodes on the same subnet and can only be sent across hubs and switches. This ensures reliable detection of link or network failures.

If you choose to route, you must ensure that your routing tables are set up so that packets intended for one network do not get routed to a different network. If this happens, there could be undetected (latent) failures which could eventually lead to system downtime due to network outage without warning. Therefore, ensure that you have independent paths to each network and that routers do not route packets between the two networks.

<SERVICE> tag

Configures the service. Features of a service include two servers associated with the service. At most, one can be active (that is, active-active is not supported). Under failover circumstances, upBeat will perform IP failover for this service if an IP address and interface are provided under the <SERVICE> tag.

A typical <SERVICE> tag might look similar to the following:

```
<SERVICE
    NAME="myService:/ipfs"
    SERVICE_ID="17"
    TYPE="BASIC"
    STARTUPDELAY_SEC="5"
    PORT="1776">
    <NODE_REF NODE_ID="1"/>
    <NODE_REF NODE_ID="2"/>
    <LINK NETWORK="Network1" ROUTE="ROUTE"/>
    <LINK NETWORK="Network2" ROUTE="ROUTE"/>
    <SCSIBEAT>
        <DISK PARTITION="system"/>
        <DISK PARTITION="one"/>
    </SCSIBEAT>
    <HANFS/>
<SERVICE_IP IP="192.168.1.1" IF="hme0:17"/>
</SERVICE>
```

upBeat optionally associates IP addresses with general services. These IP addresses migrate to the active server, and should be different from the fixed IP addresses that the operating system configures upon booting. In the example above, the active server would configure the IP address `192.168.1.1` on `hme0:17`. The standby system deconfigures `hme0:17`, if necessary. In the event of a failover, this process occurs in reverse.

NAME attribute

Required. A name for the given service; must be different from the NAME attribute setting in any of the other <SERVICE> tags. The NAME attribute should be used as a convenient file handle for other configuration files like `ipfstab`, or for system messages.

Example:

```
NAME="myService:/ipfs"
```

SERVICE_ID attribute

Required. The numerical ID of the service. Choose any number you like, as long as it is unique (among all the <SERVICE> tags) and a positive 32-bit integer greater than or equal to 1.

Example:

```
SERVICE_ID="17"
```

TYPE attribute

Required. The type of service. Currently, only one type, `BASIC`, is defined.

Example:

```
TYPE="BASIC"
```

STARTUPDELAY_SEC attribute

Required. A period of time, specified in seconds, between the time the given service is functional and the time when it is possible for upBeat to make the service active.

Example:

```
STARTUPDELAY_SEC="7"
```

PORT attribute

Optional. The port of the given service. If the port is specified, it is made available to the application to use as it sees fit.

Example:

```
PORT="1776"
```

<NODE_REF> subtag of the <SERVICE> tag

Required. One or two <`NODE_REF`> subtags are required to specify the nodes on which the service runs.

NODE_ID attribute

This tag has a single required attribute, NODE_ID, which is a node number. The number must not be the same as in any other <NODE_REF> subtag within the same <SERVICE> tag.

Example:

```
<NODE_REF NODE_ID="1"/>
```

<LINK> subtag of the <SERVICE> tag

Optional. For those services that require that the active and standby applications communicate, one or more < LINK> subtags are used to define a pair of IP addresses that the application can use to communicate. The information in the <LINK> tags is available to the application through the ubSvcIPPair() API; the application can use this information however it sees fit.

Many services do not require that the active and standby applications communicate, and for these services, the <LINK> subtag can be omitted. Also, in services with only a single node, the <LINK> subtag is not needed.

If you define links using this subtag in a <SERVICE> tag, define the same links in a <HEART-BEAT> tag so that the application can receive link callbacks for those links.

Example:

```
<LINK NETWORK="Network1" ROUTE=”ROUTE”/>
```

NETWORK attribute

Required. This allows the services to associate IP addresses from two different nodes. This is a reference to the NETWORK attribute of the <INTERFACE> subtag of the <NODE> tag.

Example:

```
NETWORK="Network1"
```

ROUTE attribute

Optional. The value of this attribute is made available to the application through the ubSvcIPPair() API; the application may use the information however it wishes. If the attribute is not present, the default behavior is to not route.

Example:

```
ROUTE="ROUTE"
```

PREF attribute

Optional. The information specified in the PREF attribute is passed to the user application to

be used in any way the application desires. For example, the PREF attribute could be used to establish the order in which the service uses the specified network. This attribute can be used to direct traffic to a private interface or to a faster interface while still having other interfaces (public and/or slower) as backups in case the most desirable link is unavailable.

If used, the link preferences on both servers must match. The PREF value must be a signed integer. The default is 0.

Example:

```
PREF="1"
```

<SCSIBEAT> subtag of the <SERVICE> tag

Optional. Can be added to any service where you need to monitor your SCSI disks more closely than Solaris allows. The <SCSIBEAT> subtag allows upBeat to be informed when one or more disks (partitions) are required for the given service to be active. If a disk is failed, a server cannot become active for the service; if a disk fails on the active server, upBeat will initiate a failover.

<DISK> subtag and PARTITION attribute

If you add the <SCSIBEAT> subtag to a service, you must use the PARTITION attribute of the <DISK> subtag to list the disk partitions from the <PARTITION> subtag of the <NODE> tag (see "<NODE> tag" on page 149).

For more information about SCSIbeat, see "SCSIbeat" on page 11.

Example:

```
<SCSIBEAT>
   <DISK PARTITION="system"/>
   <DISK PARTITION="one"/>
</SCSIBEAT>
```

<HANFS> subtag of the <SERVICE> tag

Optional. The <HANFS> subtag is used only when configuring a high availability NFS system. It has the following optional attributes:

- PORT specifies the port number of the NFS server. This is the port used by NFS clients to connect.
- NSERVER is the number of kernel threads created to handle NFS requests. This determines how many NFS requests can be handled at the same time.
- PROTOS specifies which networking protocols are configured for the NFS server. This is specified in a space-separated list. The valid protocols are UDP and TCP. Default is UDP.

Example:

```
<HANFS PORT="2049" NSERVER="17" PROTOS="UDP TCP"/>
```

Be sure you also add the required share settings to the `ipfstab` file. For more information about this and other HA NFS topics, refer to "High Availability NFS (HA NFS)" on page 107.

<SERVICE_IP> subtag of the <SERVICE> tag

Optional. This tag is used to set up the necessary information for IP failover. Therefore, this attribute can be omitted only for services that do not require IP failover. Defines the IP address, host name, and interface name of the service. A <SERVICE> tag can have more than one <SERVICE_IP> subtag, unless you are configuring for HA NFS.

When configuring for HA NFS, this subtag is required.

CAUTION: You must use one, and only one, <SERVICE_IP> subtag when configuring an HA NFS service. If any <SERVICE> tag contains an <HANFS> subtag and also contains more than one <SERVICE_IP> subtag (or does not contain a <SERVICE_IP> subtag), an error will occur and upDisk will stop running.

IP attribute

Optional. The IP address of the given service. If you use the HOST attribute, you can not use the IP attribute.

Specify a different IP address for each service.

The value of IP is passed to `inet_addr()`.

Example:

```
IP="192.168.1.1"
```

HOST attribute

Optional. The host name. Can be used instead of IP; but you can not use both. Using HOST may be convenient for testing and evaluation purposes, but we recommend that HA deployments rely on the IP attribute to reduce the reliance on unresponsive name services.

Example:

```
HOST="cpu1"
```

IF attribute

Required. The network interface on which the given service can be accessed. Specify a different interface in each <SERVICE_IP> tag. The syntax must be as follows: `hme0:`*`x`* (*x* can only be a number; no white spaces allowed).

Example:

```
IF="hme0:17"
```

Configuration File Example

Below is an example of the upSuite HA configuration file in its entirety for your reference.

```
<?xml version="1.0" ?>

<UpSuiteConfig VERSION="2">

<UPBEAT STARTUPDELAY_SEC="5"/>

<NETWORK NAME="Network1" DESCRIPTION="192.168.1.0/24"/>
<NETWORK NAME="Network2" DESCRIPTION="192.168.2.0/24"/>

<NODE NAME="left" NODE_ID="1"
 DESCRIPTION="SPARC CP1500 Solaris 2.7">
   <INTERFACE NAME="hme0" NETWORK="Network1" IP="192.168.1.2"/>
   <INTERFACE NAME="hme1" NETWORK="Network2" IP="192.168.2.2"/>
</NODE>

<NODE NAME="right" NODE_ID="2"
 DESCRIPTION="SPARC CP1500 Solaris 2.7">
   <INTERFACE NAME="hme0" NETWORK="Network1" IP="192.168.1.3"/>
   <INTERFACE NAME="hme1" NETWORK="Network2" IP="192.168.2.3"/>
</NODE>
<HEARTBEAT
   NAME="left -- right"
   TYPE="POINT_TO_POINT"
   TIMEOUT_MSEC="500"
   RESEND_MSEC="150">
   <NODE_REF NODE_ID="1"/>
   <NODE_REF NODE_ID="2"/>
   <LINK NETWORK="Network1" ROUTE=”ROUTE”/>
   <LINK NETWORK="Network2" ROUTE=”ROUTE”/>
</HEARTBEAT>
```

```
        <SERVICE
            NAME="myService:/ipfs"
            SERVICE_ID="17"
            TYPE="BASIC"
            STARTUPDELAY_SEC="5"
            PORT="1776">
            <NODE_REF NODE_ID="1"/>
            <NODE_REF NODE_ID="2"/>
            <LINK NETWORK="Network1" ROUTE="ROUTE"/>
            <LINK NETWORK="Network2" ROUTE="ROUTE"/>
            <SERVICE_IP IP="192.168.1.1" IF="hme0:17"/>
        </SERVICE>

        <SERVICE
            NAME="generalservice"
            SERVICE_ID="26"
            TYPE="BASIC"
            STARTUPDELAY_SEC="5"
            PORT="1223">
            <NODE_REF NODE_ID="1"/>
            <NODE_REF NODE_ID="2"/>
            <LINK NETWORK="Network1"/>
            <LINK NETWORK="Network2"/>
        </SERVICE>

        <SERVICE
            NAME="upstart"
            SERVICE_ID="23"
            TYPE="BASIC"
            STARTUPDELAY_SEC="5"
            PORT="1957">
            <NODE_REF NODE_ID="1"/>
            <NODE_REF NODE_ID="2"/>
            <LINK NETWORK="Network1"/>
            <LINK NETWORK="Network2"/>
        </SERVICE>

</UpSuiteConfig>
```

Sample Configurations

This section describes three example configurations (A, B, and C), including the following information for each:

- Diagram
- `upsuite.conf` file
- `/etc/ipfstab` file
- `/etc/hosts` file
- `/etc/vfstab` file
- Partition table

These configurations are provided as samples to help you configure your systems correctly. Each of these samples assumes you are running upDisk on the systems as well as upBeat.

Configuration A

In configuration A, two servers are connected via redundant networks. upBeat is running between both servers, and upDisk is running between the disks. upBeat enables the service's migrating IP address (172.17.8.177) on the server where the service is active, and disables it on the server where the service is standby.

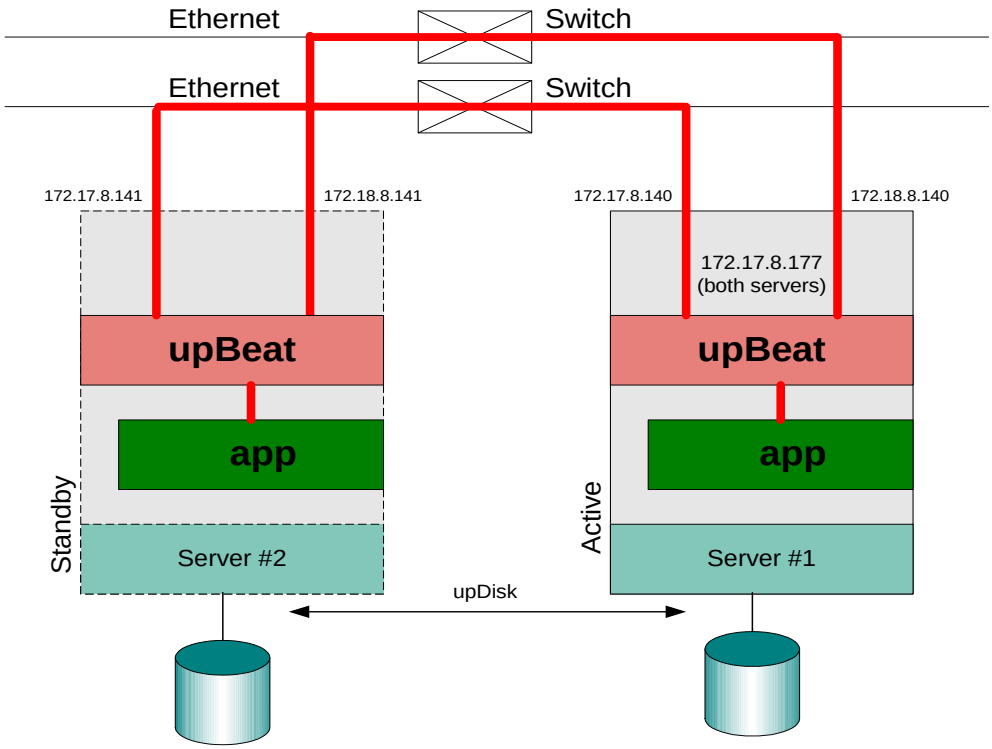

Figure 7 Configuration A block diagram

The `upsuite.conf` File

Below is the `upsuite.conf` file edited to match the system illustrated in Figure 7.

```
<?xml version="1.0" ?>

<UpSuiteConfig VERSION="2">

    <UPBEAT STARTUPDELAY_SEC="5"/>

    <NETWORK NAME="Network1"/>
    <NETWORK NAME="Network2"/>

    <NODE NAME="server1" NODE_ID="1">
           <INTERFACE NAME="hme0" NETWORK="Network1"
             IP="172.17.8.140"/>
           <INTERFACE NAME="hme1" NETWORK="Network2"
             IP="172.18.8.140"/>
    </NODE>

    <NODE NAME="server2" NODE_ID="2">
           <INTERFACE NAME="hme0" NETWORK="Network1"
             IP="172.17.8.141"/>
           <INTERFACE NAME="hme1" NETWORK="Network2"
             IP="172.18.8.141"/>
    </NODE>

    <HEARTBEAT
           NAME="server1 -- server2"
           TYPE="POINT_TO_POINT"
           TIMEOUT_MSEC="500"
           RESEND_MSEC="150">
           <NODE_REF NODE_ID="1"/>
           <NODE_REF NODE_ID="2"/>
           <LINK NETWORK="Network1"/>
           <LINK NETWORK="Network2"/>
    </HEARTBEAT>
```

```
    <SERVICE
          NAME="updisk:/ipfs"
          SERVICE_ID="17"
          TYPE="BASIC"
          STARTUPDELAY_SEC="5"
          PORT="1776">
          <NODE_REF NODE_ID="1"/>
          <NODE_REF NODE_ID="2"/>
          <LINK NETWORK="Network1"/>
          <LINK NETWORK="Network2"/>
          <SERVICE_IP IP="172.17.8.177" IF="hme0:17"/>
    </SERVICE>
</UpSuiteConfig>
```

The /etc/ipfstab File

Below is the /etc/ipfstab file edited to match the system illustrated in Figure 7.

```
# data    ipfs    ipfs    localfs   localfs   localfs    localfs              share
# set     mount   mount   mount     mount
# name    point   opts    point     type      device     opts                 opts
  /ipfs   /ipfs   -       /ipfs     ufs       c0t0d0s7   rw,noatime,logging   rw
```

The /etc/hosts File

Below is the /etc/hosts file edited to match the system illustrated in Figure 7. Note that listed to the right of the IP address is the actual name of the server.

```
#       Internet host table
#
172.17.8.177        nfs-server
172.17.8.140        server1      server1-network1
172.17.8.141        server2      server2-network1
172.18.8.140        server1-network2
172.18.8.141        server2-network2
```

The /etc/vfstab File

Below is the /etc/vfstab file edited to match the system illustrated in Figure 7.

```
#device              device               mount              FS     fsck     mount    mount
#to mount            to fsck              point              type   pass     at boot  options
#
fd                   -                    /dev/fd            fd     -        no       -
/proc                -                    /proc              proc   -        no       -
/dev/dsk/c0t0d0s1    -                    -                  swap   -        no       -
/dev/dsk/c0t0d0s0    /dev/rdsk/c0t0d0s0   /                  ufs    1        no       logging
/dev/dsk/c0t0d0s6    /dev/rdsk/c0t0d0s6   /usr               ufs    1        no       logging
/dev/dsk/c0t0d0s3    /dev/rdsk/c0t0d0s3   /var               ufs    1        no       logging
/dev/dsk/c0t0d0s4    /dev/rdsk/c0t0d0s4   /opt               ufs    2        yes      logging
swap                 -                    /tmp               tmpfs  -        yes      -
#This partition is reserved for upDisk. Leave it commented out.
#/dev/dsk/c0t0d0s7   /dev/rdsk/c0t0d0s7   -                  -      -        -        -
```

Configuration B

Like configuration A, configuration B features two servers connected via redundant networks. upBeat is running between both servers; upBeat enables one IP address (172.17.8.177) on the server where the service is currently active. In addition, an array of disks is attached to each server via a metadevice managed by Solaris' Online Disk Suite. upDisk is replicating the file systems on the metadevice.

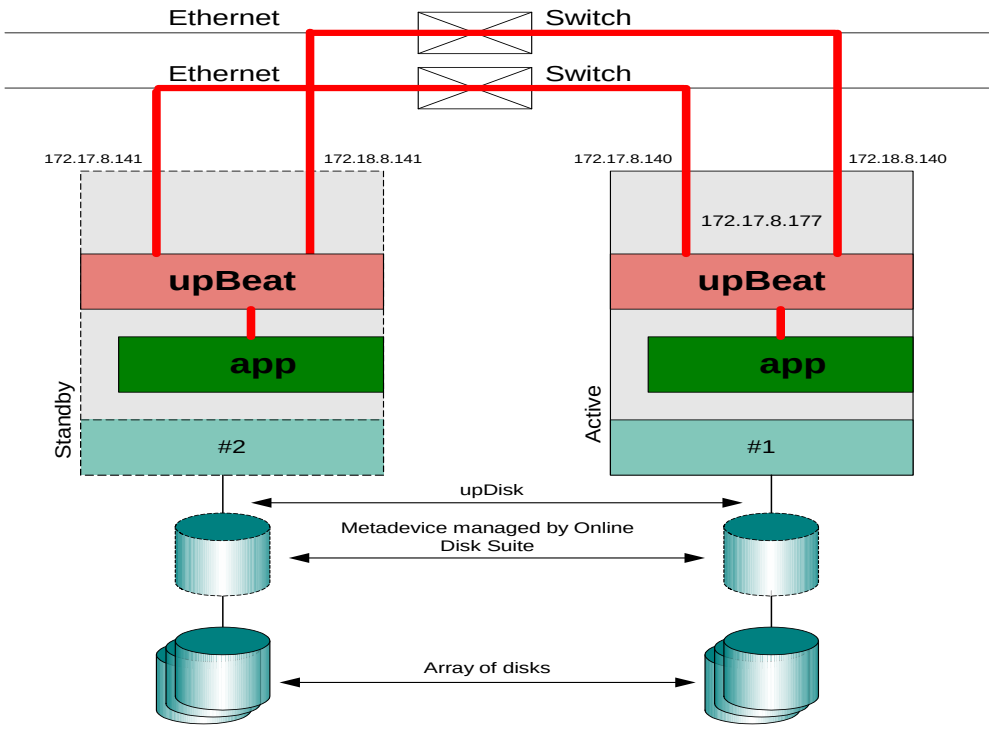

Figure 8 Configuration B block diagram

The `upsuite.conf` File

Below is the `upsuite.conf` file edited to match the system illustrated in Figure 8.

```
<?xml version="1.0" ?>

<UpSuiteConfig VERSION="2">

       <UPBEAT STARTUPDELAY_SEC="5"/>

       <NETWORK NAME="Network1"/>
       <NETWORK NAME="Network2"/>

       <NODE NAME="server1" NODE_ID="1">
              <INTERFACE NAME="hme0" NETWORK="Network1"
                IP="172.17.8.140"/>
              <INTERFACE NAME="hme1" NETWORK="Network2"
                IP="172.18.8.140"/>
       </NODE>

       <NODE NAME="server2" NODE_ID="2">
              <INTERFACE NAME="hme0" NETWORK="Network1"
                IP="172.17.8.141"/>
              <INTERFACE NAME="hme1" NETWORK="Network2"
                IP="172.18.8.141"/>
       </NODE>

       <HEARTBEAT
              NAME="server1 -- server2"
              TYPE="POINT_TO_POINT"
              TIMEOUT_MSEC="500"
              RESEND_MSEC="150">
              <NODE_REF NODE_ID="1"/>
              <NODE_REF NODE_ID="2"/>
              <LINK NETWORK="Network1"/>
              <LINK NETWORK="Network2"/>
       </HEARTBEAT>
```

```
        <SERVICE
                NAME="updisk:/ipfs"
                SERVICE_ID="17"
                TYPE="BASIC"
                STARTUPDELAY_SEC="5"
                PORT="1776">
                <NODE_REF NODE_ID="1"/>
                <NODE_REF NODE_ID="2"/>
                <LINK NETWORK="Network1"/>
                <LINK NETWORK="Network2"/>
                <SERVICE_IP IP="172.17.8.177" IF="hme0:17"/>
        </SERVICE>

</UpSuiteConfig>
```

The `/etc/ipfstab` File

Below is the `/etc/ipfstab` file edited to match the system illustrated in Figure 8.

```
# data    ipfs     ipfs    localfs   localfs  localfs        localfs              share
# set     mount    mount   mount     mount
# name    point    opts    point     type     device         opts                 opts
  /ipfs   /ipfs    -       /ipfs     ufs      d0             rw,noatime,logging   rw
```

The /etc/hosts File

Below is the /etc/hosts file edited to match the system illustrated in Figure 8. Note that listed to the right of the IP address is the actual name of the server.

```
#       Internet host table
#
172.17.8.177        NFS server
172.17.8.140        server1      server1-network1
172.17.8.141        server2      server2-network1
172.18.8.140        server1-network2
172.18.8.141        server2-network2
```

The /etc/vfstab File

Below is the /etc/vfstab file edited to match the system illustrated in Figure 8.

```
#device            device               mount       FS      fsck  mount    mount
#to mount          to fsck              point       type    pass  at boot  options
#
fd                 -                    /dev/fd     fd      -     no      -
/proc              -                    /proc       proc    -     no      -
/dev/dsk/c0t0d0s1  -                    -           swap    -     no      -
/dev/dsk/c0t0d0s0  /dev/rdsk/c0t0d0s0   /           ufs     1     no      logging
/dev/dsk/c0t0d0s6  /dev/rdsk/c0t0d0s6   /usr        ufs     1     no      logging
/dev/dsk/c0t0d0s3  /dev/rdsk/c0t0d0s3   /var        ufs     1     no      logging
/dev/dsk/c0t0d0s4  /dev/rdsk/c0t0d0s4   /opt        ufs     2     yes     logging
swap               -                    /tmp        tmpfs   -     yes     -
# These partitions are reserved for OLDS metadevice /dev/md/dsk/d0.
#/dev/dsk/c0t1d0s2 /dev/rdsk/c0t1d0s2   -           -       -     -        -
#/dev/dsk/c0t2d0s2 /dev/rdsk/c0t1d0s2   -           -       -     -        -
#/dev/dsk/c0t3d0s2 /dev/rdsk/c0t1d0s2   -           -       -     -        -

#This partition is reserved for upDisk. Leave it commented out.
#/dev/md/dsk/do    /dev/md/rdsk/d0      /ipfs       ufs     2     yes      noatime,logging
```

Configuration C

In configuration C, two servers are connected via redundant networks. upBeat is running between both servers, and upDisk is running between the disks. The upDisk service uses one migrating IP address (172.17.8.177). Two clients are connected to the network via redundant links and are running upBeat.

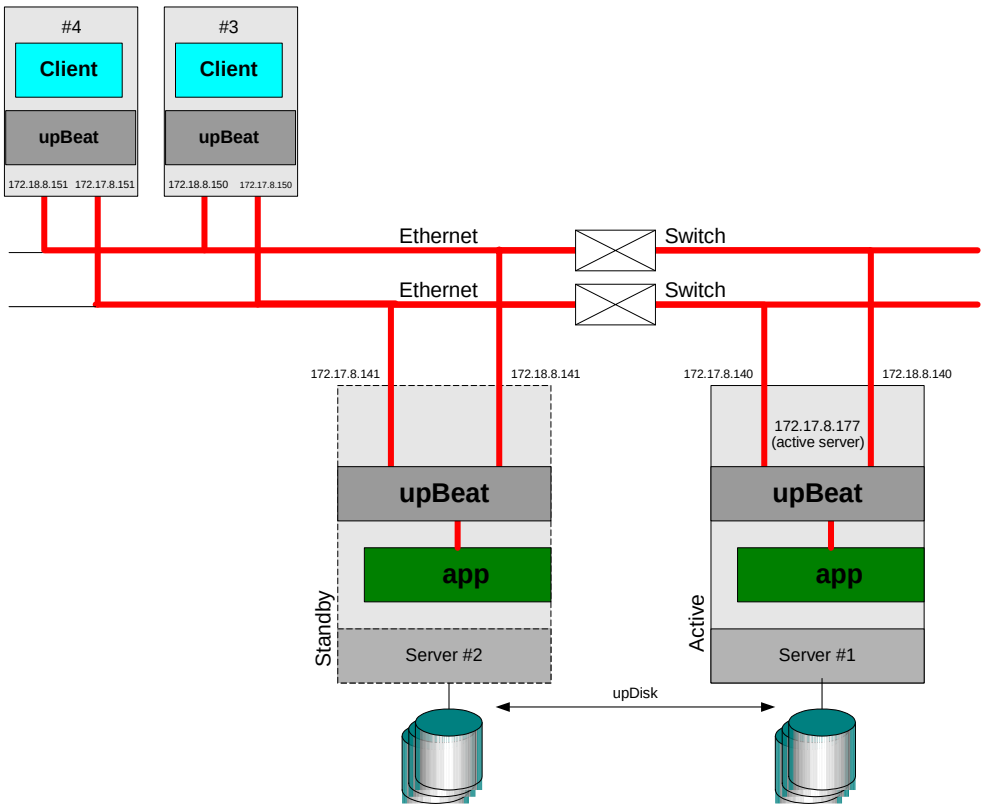

Figure 9 Configuration C block diagram

The `upsuite.conf` File

Below is the `upsuite.conf` file edited to match the system illustrated in Figure 9.

Note: If your clients are separated from your servers by routers (e.g., LANs, MANs, or WANs) you will need to enable the routing feature in the configuration file. The default is to not route, which is how the sample file below is configured; notice that ROUTE does not appear beneath the <HEARTBEAT> and <SERVICE> tags. For information about how to enable routing, see "<HEARTBEAT> tag" on page 152 and "<SERVICE> tag" on page 154.

```
<?xml version="1.0" ?>

<UpSuiteConfig VERSION="2">

        <UPBEAT STARTUPDELAY_SEC="5"/>

        <NETWORK NAME="NetworkA"/>
        <NETWORK NAME="NetworkB"/>

        <NODE NAME="server1" NODE_ID="1">
               <INTERFACE NAME="hme0" NETWORK="NetworkA" IP="172.17.8.140"/>
               <INTERFACE NAME="hme1" NETWORK="NetworkB" IP="172.18.8.140"/>
        </NODE>

        <NODE NAME="server2" NODE_ID="2">
               <INTERFACE NAME="hme0" NETWORK="NetworkA" IP="172.17.8.141"/>
               <INTERFACE NAME="hme1" NETWORK="NetworkB" IP="172.18.8.141"/>
        </NODE>

        <NODE NAME="client1" NODE_ID="3">
               <INTERFACE NAME="hme0" NETWORK="NetworkA" IP="172.17.8.150"/>
               <INTERFACE NAME="hme1" NETWORK="NetworkB" IP="172.18.8.150"/>
        </NODE>

        <NODE NAME="client2" NODE_ID="4">
               <INTERFACE NAME="hme0" NETWORK="NetworkA" IP="172.17.8.151"/>
               <INTERFACE NAME="hme1" NETWORK="NetworkB" IP="172.18.8.151"/>
        </NODE>
```

```
<HEARTBEAT
       NAME="server1 -- server2"
       TYPE="POINT_TO_POINT"
       TIMEOUT_MSEC="500"
       RESEND_MSEC="150">
       <NODE_REF NODE_ID="1"/>
       <NODE_REF NODE_ID="2"/>
       <LINK NETWORK="NetworkA"/>
       <LINK NETWORK="NetworkB"/>
</HEARTBEAT>
<HEARTBEAT
       NAME="3 -- 1"
       TYPE="POINT_TO_POINT"
       TIMEOUT_MSEC="500"
       RESEND_MSEC="150">
       <NODE_REF NODE_ID="3"/>
       <NODE_REF NODE_ID="1"/>
       <LINK NETWORK="NetworkA"/>
       <LINK NETWORK="NetworkB"/>
</HEARTBEAT>
<HEARTBEAT
       NAME="4 -- 1"
       TYPE="POINT_TO_POINT"
       TIMEOUT_MSEC="500"
       RESEND_MSEC="150">
       <NODE_REF NODE_ID="4"/>
       <NODE_REF NODE_ID="1"/>
       <LINK NETWORK="NetworkA"/>
       <LINK NETWORK="NetworkB"/>
</HEARTBEAT>
<HEARTBEAT
       NAME="3 -- 2"
       TYPE="POINT_TO_POINT"
       TIMEOUT_MSEC="500"
       RESEND_MSEC="150">
       <NODE_REF NODE_ID="3"/>
       <NODE_REF NODE_ID="2"/>
       <LINK NETWORK="NetworkA"/>
       <LINK NETWORK="NetworkB"/>
</HEARTBEAT>
```

```
        <HEARTBEAT
                NAME="4 -- 2"
                TYPE="POINT_TO_POINT"
                TIMEOUT_MSEC="500"
                RESEND_MSEC="150">
                <NODE_REF NODE_ID="4"/>
                <NODE_REF NODE_ID="2"/>
                <LINK NETWORK="NetworkA"/>
                <LINK NETWORK="NetworkB"/>
        </HEARTBEAT>
        <SERVICE
                NAME="updisk:/ipfs"
                SERVICE_ID="17"
                TYPE="BASIC"
                STARTUPDELAY_SEC="5"
                PORT="1776">
                <NODE_REF NODE_ID="1"/>
                <NODE_REF NODE_ID="2"/>
                <LINK NETWORK="NetworkA"/>
                <LINK NETWORK="NetworkB"/>
                <SERVICE_IP IP="172.17.8.177" IF="hme0:17"/>
        </SERVICE>
</UpSuiteConfig>
```

The `/etc/ipfstab` File

Below is the `/etc/ipfstab` file edited to match the system illustrated in Figure 9.

```
# data   ipfs    ipfs    localfs    localfs   localfs      localfs                share
# set    mount   mount   mount      mount
# name   point   opts    point      type      device       opts                   opts
  /ipfs  /ipfs   -       ipfs       ufs       c0t0d0s7     rw,noatime,logging     rw
```

The /etc/hosts File

Below is the /etc/hosts file edited to match the system illustrated in Figure 9. Note that listed to the right of the IP address is the actual name of the server.

```
#       Internet host table
#
172.17.8.177        NFS server
172.17.8.140        server1      server1-network1
172.17.8.141        server2      server2-network1
172.18.8.140        server1-network2
172.18.8.141        server2-network2
client1
172.17.8.150        client1a
172.18.8.150        client1b
client2
172.17.8.151        client2a
172.18.8.151        client2b
```

The /etc/vfstab File

Below is the /etc/vfstab file edited to match the system illustrated in Figure 9.

```
#device              device              mount             FS    fsck    mount   mount
#to mount            to fsck             point             type  pass    at boot options
#
fd                   -                   /dev/fd           fd    -        no     -
/proc                -                   /proc             proc  -        no     -
/dev/dsk/c0t0d0s1    -                   -                 swap  -        no     -
/dev/dsk/c0t0d0s0    /dev/rdsk/c0t0d0s0  /                 ufs   1        no     logging
/dev/dsk/c0t0d0s6    /dev/rdsk/c0t0d0s6  /usr              ufs   1        no     logging
/dev/dsk/c0t0d0s3    /dev/rdsk/c0t0d0s3  /var              ufs   1        no     logging
/dev/dsk/c0t0d0s4    /dev/rdsk/c0t0d0s4  /opt              ufs   2        yes    logging
swap                 -                   /tmp              tmpfs -        yes    -
#This partition is reserved for upDisk. Leave it commented out.
#/dev/dsk/c0t0d0s7   /dev/rdsk/c0t0d0s7  -                 -     -        -      -
```

Partition Table

Table 12 defines the partition table from the systems illustrated in Figure 7, Figure 8, and Figure 9.

Note: For sample configuration B (Figure 8), the system disk's partitions are illustrated in Table 12. The disks used by Online Disk Suite would have different partition tables in which only partition 2, most likely, would be configured.

Part	Tag	Flag	Cylinders	Size	Blocks
0	root	wm	0 - 327	512.50MB	(328/0/0) 1049600
1	swap	wu	328 - 655	512.50MB	(328/0/0) 1049600
2	backup	wm	0 - 11198	17.09GB	(11199/0/0) 35836800
3	var	wm	656 - 983	512.50MB	(328/0/0) 1049600
4	unassigned	wm	985 - 1640	1.00GB	(656/0/0) 2099200
5	unassigned	wu	0	0	(0/0/0) 0
6	usr	wm	1641 - 2296	1.00GB	(656/0/0) 2099200
7	unassigned	wm	2297 - 11198	13.58GB	(8902/0/0) 28486400

Table 12 Partition table for sample configurations A, B, and C

Engineering Guidelines for UpSuite

This section describes the configuration settings that are required for an improvement in the performance of Upsuite.

The sync rate of files from active server to standby server depends on the following parameters:

1. Disk Latency
2. TCP/IP Network Speed
3. Physical Memory
4. /etc/ipfstab options
5. CPU speed

The memory used by upSuite depends on the number of datasets used, the number of files created and their sizes and the options in /etc/ipfstab.

The number of files and the sizes of files upSuite can replicate depends on limit imposed by UFS.

The number of datasets upSuite supports is 262143 (0x3ffff is the limit in Solaris on minor number for device drivers)

The below specified options can be used to improve the performance of upSuite.

ipfstab File

The following options can be set in /etc/ipfstab file for an improvement in the performance of upSuite on Solaris U3 and U5 based on the number of data sets used and the size of replicated data.

1. maxmem
2. throttle
3. maxops

For example, a system, which has 20 datasets and has more than 150GB of data for replication, can have the following entry in the /etc/ipfstab file with maxmem and throttle set.

```
# data   ipfs    ipfs    localfs    localfs   localfs      localfs                share
# set    mount   mount   mount      mount
# name   point   opts    point      type      device       opts                   opts
  /ipfs  /ipfs maxmem=3g,throttle=95:100ipfs ufs c0t0d0s7 rw,noatime,loggingrw
```

TCP/IP Configuration

The TCP maximum buffer size, the congestion window size, the transmit buffer size and the the receive buffer size can be set for improving the perfomance of upSuite on Solaris 10 U3 and U5 as per Solaris TCP/IP tuning guidelines.

For example, for a 1Gbps link, the following commands can be used to set the above specified parameters.

```
ndd -set /dev/tcp tcp_max_buf 16777216

ndd -set /dev/tcp tcp_cwnd_max 8388608

ndd -set /dev/tcp tcp_xmit_hiwat 1048576

ndd -set /dev/tcp tcp_recv_hiwat 1048576
```

Part III: ubManager

ubManager is a software layer situated above upBeat, the heartbeat manager of upSuite. Where upBeat detects link, network, and node failures, ubManager extends this functionality with its ability to:

- Start, stop, and monitor non-HA-aware user application processes
- Group services and coordinate failover among them
- Add control through programs and scripts
- Monitor resources

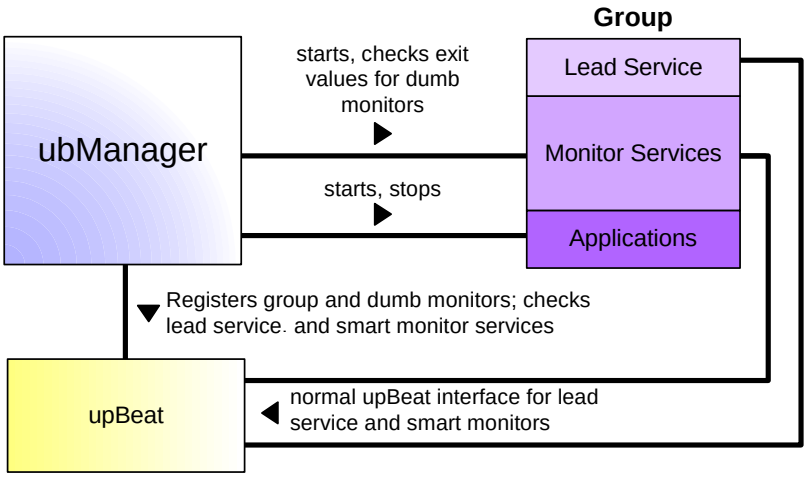

Figure 1 ubManager and upBeat

15 Introduction to ubManager

ubManager™ (short for “upBeat Manager” and pronounced “You-Be-Manager”) is designed to integrate non-HA-aware user applications into the upSuite™ HA framework.

This guide provides the following information related to ubManager:

- Description of features
- Command reference
- ubManager Monitors

We assume that as a user of ubManager, you are familiar with:

- The Solaris operating system
- TCP/IP and networking
- XML Notation
- Continuous Computing Corporation’s upBeat failure detection management software
- Continuous Computing Corporation’s upDisk file system replication software

In this chapter

What is ubManager? 179
How Does ubManager Work? 180

What is ubManager?

ubManager is a software layer situated above upBeat, the heartbeat manager of upSuite. Where upBeat detects link, network, and node failures, ubManager extends this functionality with its ability to:

- Start, stop, and monitor non-HA-aware user application processes
- Group services and coordinate failover among them
- Add control through programs and scripts
- Monitor resources

How Does ubManager Work?

Figure 1 illustrates the relationship between ubManager, a ubManager service group, and upBeat.

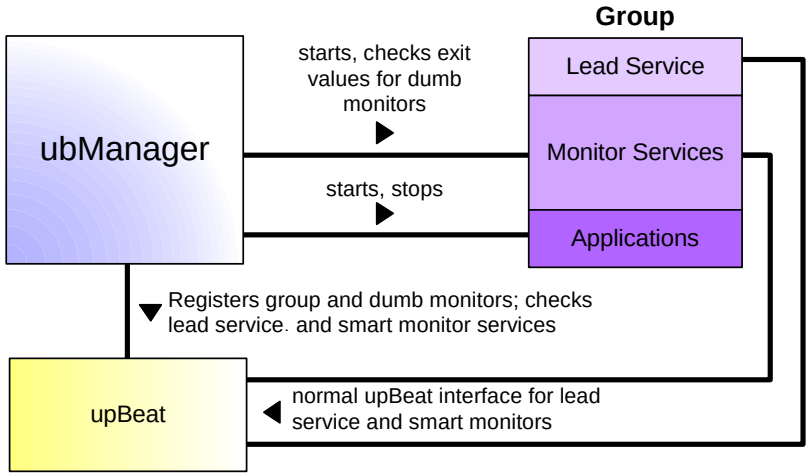

Figure 1 ubManager and upBeat

Service Groups

By default, each upBeat service is independent of all others. However, in a typical implementation, certain dependencies exist among upBeat services and your applications. For example, your database application might depend on upDisk, upSuite's data replication module, for the ability to write to an upDisk dataset. If dependencies such as these exist between services, you would need to implement special policy software to manage those relationships. ubManager is that software.

ubManager allows you to create groups of services controlled by a "lead service" and/or "monitors" (see below) which you designate. All services associated with the lead service follow its behavior in the event of a change to its status. For example, if an upDisk dataset is the lead service for a database application and the dataset fails due to a disk failure, the database application also fails over and so do all other application members of the group specifying that particular upDisk dataset as the lead service.

A ubManager service group has the following characteristics:

- **<SERVICE> Entry in upSuite Configuration File.**
  Each ubManager service group is itself an upBeat service, which ubManager controls, so each group must have its own <SERVICE> tag entry in the upSuite configuration file, `/etc/upsuite/upsuite.conf`.

- **Lead Service (optional).**
  ubManager controls the group to follow any designated lead service, with exceptions noted below. This means that when the lead service becomes active, ubManager registers the group with upBeat to be active, and when the lead service goes to standby, ubManager registers the group with upBeat to be standby. Since ubManager passively tracks the lead service, any service chosen as a group lead must already be independently integrated with upBeat. If a user application uses disk storage, an upDisk dataset would serve as an ideal lead service.

  If a group has no lead service, then upon initialization ubManager will register the group with upBeat to be active, dependent upon the status of any monitors. If the group's peer is already active, then upBeat will issue a standby directive for the group. The `ubmfailover` utility can be used to failover a group whether it has a lead service or not.

- **User Applications.**
  These are application programs that are non-upBeat aware. When a group becomes active on a node, ubManager starts the user applications for that group on that node, and when a group becomes standby on a node, ubManager gracefully terminates the user applications for that group on that node.

- **Failover Scripting.**
  After ubManager receives a directive from upBeat, it runs either a "go active" script (for an active directive) or a "go standby" script (for a standby directive) before acknowledging the directive. These scripts provide failover scripting, and can launch or terminate processes, configure network interfaces, or manage any other configuration for the group.

- **Monitors (optional).**
  A group can be configured to use processes that monitor resources required by the group. These processes are called monitors and are themselves upBeat services. If any monitor for a group indicates that the resource it's monitoring is "unhealthy," ubManager will failover the group, provided that all the monitors for the peer group on the peer node indicate their resources are "healthy."

Resource Monitoring

A ubManager group can be configured to use a number of monitors to monitor the health of the resources required by the group, and so determine whether a group can become or can continue to be active. Monitors are upBeat services. However, monitors on one node are independent of monitors on other nodes, they do not have upSuite peers; here, upBeat is only used as an event manager, informing ubManager of the state of peer groups' monitors. Monitors may be shell scripts or application binaries, they may run in a transient or permanent manner, and they may have their own independent interface with upBeat or have ubManager handle that interface for them.

Each monitor is designed to watch a particular element of the system and report status back to ubManager either by the monitor's exit value or through upBeat. If any one monitor for a group indicates that the resource it's monitoring is unhealthy, then ubManager will failover a group if all of the monitors for the peer group indicate that the peer group's resources are healthy; if the peer group cannot become active, then the local group will remain active.

ubManager can support two types of monitor: "smart" or "dumb." Smart monitors are able to directly update their service status with upBeat, while dumb monitors rely on ubManager to do this for them. When a smart monitor is active, that informs ubManager that the resource that the monitor is monitoring is healthy. Likewise, when a smart monitor is standby, that informs ubManager that the resource that the monitor is monitoring is unhealthy.

Dumb monitors may be permanent or periodic. When a dumb permanent monitor exits (no matter its exit value), that informs ubManager that the resource that the monitor is monitoring is unhealthy, and ubManager registers the monitor's upBeat service as standby (if it is not already) and restarts the monitor. A dumb periodic monitor is expected to exit quickly and ubManager restarts it at a specified interval. The exit value of a dumb periodic monitor indicates if the resource that it's monitoring is healthy (exit value of zero) or unhealthy (non-zero exit value).

Monitor processes may be scripts or binary executables. Scripting can provide a simple, yet powerful, way to implement a monitor, where disk usage, processor utilization, process status, and the like can be monitored. "ubManager Monitors" on page 203 contains information about the monitors shipped with ubManager. As an example, below is loadMon, a dumb periodic monitor, which monitors CPU load:

```
#!/bin/sh
# load-mon is invoked with a 15 minute average load limit and exits with
# 0 (success) if the current 15 minute average load is below the passed-
# in limit, or 1 (error) if the current 15 minute average load is above
# the limit.
```

```
if [ $# -ne 1 ]; then
     echo "usage: $0 <limit, eg. 0.80>"
     exit 1
fi

# Extract the load average over the past 15 minutes from uptime.
# Multiply by 100 to make the value an integer.
UPTIME=`uptime | awk -e '{print int(substr($10,1)*100)}'`

# Multiply the user's passed-in load limit by 100 to allow us to compare
# it with the current load.
LIMIT=`echo $1 | awk -e '{ print int($1*100) }'`

echo "ut=$UPTIME ul=$LIMIT"

if [ $UPTIME -lt $LIMIT ]; then
     # The load is below the limit, return success (0)
     exit 0
else
     # The load is above the limit, return error (1)
     exit 1
fi
```

Application Process Launching and Monitoring

A user application process is started when the group it belongs to becomes active, and gracefully terminated when the group becomes standby. In addition, if an application process terminates while the group is active, a failover will be triggered.

User application processes can be scripts or compiled programs.

ubManager terminates user applications by sending them a SIGTERM signal. In order to shut down gracefully, user applications should catch this signal and when received perform any termination activities (e.g., deleting temporary files) and exit within 10 seconds. If an application has not exited by that time, ubManager will forcefully terminate it with a SIGKILL signal.

Failover Scripting

User-definable shell scripts are called by ubManager during failover transitions.

These scripts provide failover scripting, and can launch or terminate processes, configure network interfaces, or manage any other configuration for the group.

- `goactive`
  After ubManager registers a group with upBeat to be active and has received a directive from upBeat for the group to go active, ubManager runs the `goactive` script. It is

passed the group name as an argument, so the same script may be used for all groups. If the script exits with a value of zero, ubManager starts all of the user applications of the group and acknowledges the directive. If the `goactive` script does not exist or if it exits with a non-zero value, then ubManager does not start the user applications and acknowledges the directive with a standby status for the group, thus preventing the group from going active.

- `gostandby`
  After ubManager registers a group with upBeat to be standby and has received a directive from upBeat for the group to go standby, ubManager acknowledges the directive, then shuts down the user applications and runs the `gostandby` script. It is passed the group name as an argument, so the same script may be used for all groups. The standby directive cannot be denied like the active directive can, so the exit value of the `gostandby` script is ignored.
- `cause_failover`
  This script is called when a group with a lead service is to failover because `ubmfailover` has been run by the operator, a monitor within the group indicates an unhealthy resource, or an application within the group unexpectedly exited. It is passed the group name as an argument and is expected to run whatever commands are needed to initiate a failover of the lead service.

ubManager Components

Figure 2 on the next page illustrates ubManager's components.

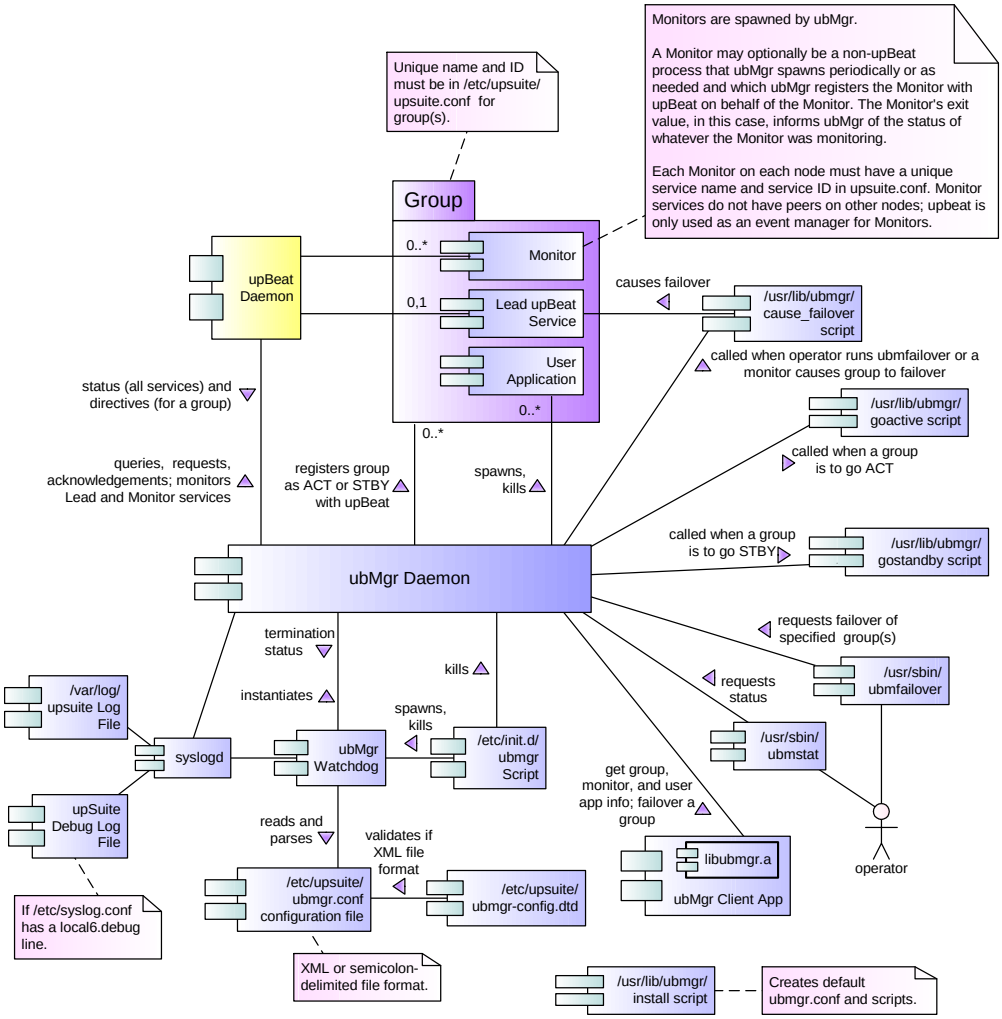
Monitors are spawned by ubMgr.
A Monitor may optionally be a non-upBeat process that ubMgr spawns periodically or as needed and which ubMgr registers the Monitor with upBeat on behalf of the Monitor. The Monitor's exit value, in this case, informs ubMgr of the status of whatever the Monitor was monitoring.
Each Monitor on each node must have a unique service name and service ID in upsuite.conf. Monitor services do not have peers on other nodes; upbeat is only used as an event manager for Monitors.
Unique name and ID must be in /etc/upsuite/upsuite.conf for group(s).
Group
0..*
Monitor
0,1
Lead upBeat Service
User Application
0..*
upBeat Daemon
causes failover
/usr/lib/ubmgr/cause_failover script
called when operator runs ubmfailover or a monitor causes group to failover
/usr/lib/ubmgr/goactive script
called when a group is to go ACT
status (all services) and directives (for a group)
queries, requests, acknowledgements; monitors Lead and Monitor services
registers group as ACT or STBY with upBeat
spawns, kills
0..*
called when a group is to go STBY
/usr/lib/ubmgr/gostandby script
ubMgr Daemon
requests failover of specified group(s)
/usr/sbin/ubmfailover
termination status
kills
instantiates
requests status
/usr/sbin/ubmstat
/var/log/upsuite Log File
spawns, kills
syslogd
ubMgr Watchdog
/etc/init.d/ubmgr Script
get group, monitor, and user app info; failover a group
upSuite Debug Log File
reads and parses
validates if XML file format
libubmgr.a
ubMgr Client App
operator
/etc/upsuite/ubmgr.conf configuration file
/etc/upsuite/ubmgr-config.dtd
If /etc/syslog.conf has a local6.debug line.
XML or semicolon-delimited file format.
/usr/lib/ubmgr/install script
Creates default ubmgr.conf and scripts.

Figure 2 ubManager components

16 ubManager Command Reference

This chapter explains ubManager's commands.

Shell Commands

This section describes the usage and function of ubManager's UNIX shell commands. Commands are listed in alphabetical order.

NAME

ubmfailover

USAGE

/usr/sbin/ubmfailover [-f] [-h *hostname*] [-p *port*] group...

DESCRIPTION

This command informs ubManager that the operator wants the specified group(s) to failover. If a called-for group has a lead service, ubManager calls the `cause_failover` script, with the name of the group as an argument, that is expected to cause the lead service for the group to failover. If the group does not have a lead service, ubManager immediately registers the group for standby if it is active.

The primary use of **`ubmfailover`** is to cause an active group to go to standby. If there is no lead service, then running **`ubmfailover`** with the name of a group in standby will have no effect. If a group's lead service is an upDisk dataset, then it is possible to cause a standby dataset to be active, thus causing the group to become active, by having the `cause_failover` script call the **`udactive`** command with **`-fA`** options for the standby dataset. If a lead service other than upDisk is used, then a communication mechanism other than upBeat must be used between the standby and active service instances to make the standby service active via the `cause_failover` script.

Note: The default `cause_failover` script must be modified to include the proper handling for each group.

OPTIONS

`-f`

This option includes the “force” option when calling the `cause_failover` script. Any special action taken by this option is up to the `cause_failover` script.

`-h` ***`hostname`***

This option changes the IP address that **`ubmfailover`** uses to communicate with ubManager (IPv4 IP addresses only are supported). The default is the local host. ***`hostname`*** is the resolvable name of a machine, not its IP address. Note that no form of security is used (e.g., authentication) with this command, and it can be run by root and non-root users.

`-p` ***`port`***

This option changes the TCP port used to communicate with ubManager. The default is 2005.

`-?` (or **`*`**)

This option prints help for the command usage and exits.

FILES

`cause_failover`

SEE ALSO

None

NAME

ubmgr (init.d)

USAGE

/etc/init.d/ubmgr {start | stop | restart}

DESCRIPTION

ubManager is normally started by an rc script (/etc/rc3.d/S99ubmgr) during system startup and terminated by another rc script (/etc/rc2.d/K99ubmgr) during system shutdown. The **/etc/init.d/ubmgr** command allows you to stop, start, or restart ubManager manually.

OPTIONS

start	Starts ubManager.
stop	Stops ubManager
restart	Restarts ubManager.

FILES

ubmgr.conf

SEE ALSO

ubmgr (admin command)

NAME

ubmgr (admin command)

USAGE

```
/usr/sbin/ubmgr [-c control_port] [-d]
     [-f config_file] [-p client_port]
     [-s script_dir] [-S] [-v] [-?]
```

DESCRIPTION

This command runs ubManager. To run ubManager with non-default options, the init script may be edited, or ubManager may be started manually from the shell prompt.

OPTIONS

-c *control_port*

ubManager uses a TCP port to determine whether or not it is the only ubManager instance running. This option allows this "control port" to be changed. The default is 2006.

-d

This option runs ubManager in the foreground (not as a daemon). All output is sent to `stdout` instead of `syslog`. Useful for debugging.

-f *config_file*

This option causes ubManager to read the configuration file you specify here rather than its default configuration file, `/etc/upsuite/ubmgr.conf`.

-p client_*port*

This option changes the TCP port that clients (`ubmstat`, `ubmfailover`) use to communicate with ubManager. The default is 2005.

-s *script_dir*

This option changes the directory that ubManager will look for the user-definable scripts (`goactive`, `gostandby`, `cause_failover`). The default is `/usr/lib/ubmgr`.

-S

This option causes program messages that would normally be sent to the syslog to instead be sent to the console or to the terminal that started ubmgr.

-v

This option prints the ubManager version number and exits.

OPTIONS cont'd	**`-?`** (or **`*`**) `This option prints help for the command usage and exits.`
FILES	`ubmgr.conf`
SEE ALSO	`ubmgr` (init.d)

NAME

ubmstat

USAGE

/usr/sbin/ubmstat [-a] [-h *hostname*]
[-i *interval*] [-m] [-N] [-p *port*] [-v] [-?]

DESCRIPTION

This command outputs various status information about ubManager groups, monitors, and applications. The basic status output is state information.

For a group, the states are UNREG (unregistered with upBeat), WAIT_MON (a group is waiting for its monitors to start), WAIT_ACT (a group has registered for active and is waiting for an active directive), WAIT_STBY (a group has registered for standby and is waiting for a standby directive), INIT_ACT (a group has received an active directive and is waiting for all of its applications to start), INIT_STBY (a group has received a standby directive and is waiting for its applications to terminate), ACT (a group is active, its applications are running), STBY (a group is standby, its applications are not running), and THRASH (a group has registered for standby or active a number of times but has not yet received a corresponding directive).

For a monitor, the states are UNREG (unregistered with upBeat), SMART (the monitor is running and interfaces to upBeat itself; use the **ups** command to find its upBeat service state), WAIT_ACT (ubManager has registered the monitor to be active and is waiting for an active directive for the monitor), WAIT_STBY (ubManager has registered the monitor to be standby and is waiting for a standby directive for the monitor), ACT (the monitor is active, and is therefore indicating that the resource it's monitoring is healthy), STBY (the monitor is standby, and is therefore indicating that the resource it's monitoring is not healthy), REFUSE_ACT (ubManager has started a monitor and registered that monitor to be active, but before receiving an active directive the monitor has terminated), and WAIT_STBY_REREG (ubManager has registered a monitor to be standby and while waiting for a standby directive for the monitor, the monitor has become active).

For an application, the states are DEAD (the application is not running), RUN (the application is running), and BROKEN (the application has been started and has unexpectedly died a number of times).

OPTIONS

-a

This option includes the application status in the output.

-h *hostname*

This option changes the IP address that **ubmstat** uses to communicate with ubManager (IPv4 IP addresses only are supported). The default is the local host. ***hostname*** is the resolvable name of a machine, not its IP address. Note that no form of security is used (e.g., authentication) with this command, and it can be run by root and non-root users.

-i *interval*

If this option is specified, **ubmstat** will not exit until you send a break with **Ctrl-C**. Each ***interval*** seconds, **ubmstat** prints out the latest status.

-m

This option includes the monitor status in the output.

-N

This option will cause group status to not be in the output. If **-N** is used but neither **-a** nor **-m**, then **ubmstat** will behave as if all three options were used.

Example

```
# ubmstat -ma

Group     State
--------------
group1    ACT
group2    STBY

Monitor             State
------------------------
g1n1m1              ACT
g1n1m2              SMART
g1n2m1              UNREG
g1n2m2              UNREG
g2n1m1              ACT
g2n1m2              ACT
g2n2m1              UNREG
g2n2m2              UNREG

Appl                State
------------------------
ubapp11             RUN
ubapp12             RUN
ubapp21             DEAD
ubapp22             DEAD
ubapp23             DEAD
```

-p *port*

This option changes the TCP port used to communicate with ubManager. The default is 2005.

-v

This option, short for "verbose," provides additional informational output, often helpful for troubleshooting.

OPTIONS (cont.)

For group information:

For all output except otherwise indicated:

`1` = active

`0` = standby

`Gs/Gp`

Indicates the group's state on the self (local) node (`Gs`) and the peer node (`Gp`).

`Ts/Tp`

Indicates the group's lead (or tracking) service's state on the self node (`Ts`) and the peer node (`Tp`).

`Ms/Mp`

Overall aggregate state of all the group's monitors on self (`Ms`) and peer (`Mp`) nodes.

`AppOk`

`1` indicates that all of the applications in the group are initializing and/or running on the self node.

`0` indicates that one or more applications in the group are not running and/or are in the process of terminating on the self node.

`AppDead`

`1` indicates that all applications in the group are dead on the self node.

`0` indicates one or more applications in the group are running on the self node.

Example

```
# ubmstat -v
Group     State [Gs/Gp] [Ts/Tp] [Ms/Mp] AppOk AppDead
-------------------------------------------------------
group1    ACT   [ 1/0 ] [ 1/0 ] [ 1/1 ]     1       0
group2    STBY  [ 0/1 ] [ 0/1 ] [ 1/1 ]     0       1
```

OPTIONS (cont.)

For monitor information:

`Ms/Mp`

State of self (`Ms`) and peer (`Mp`) monitor (1 = active, 2 = non-existent).

`PSt`

Process state of monitor (DEAD, RUN, or BROKEN). For a dumb periodic monitor, it is possible that the monitor state is active, but that the process itself is DEAD until ubManager restarts it.

`PID`

Last process ID of monitor.

`Exit`

Last exit value of monitor.

`Num Times Forked`

The number of times that the monitor has been instantiated.

`Num Times Died`

The number of times that the monitor has unexpectedly terminated.

`Num Times AWOL`

The number of times that the monitor has been sent a termination signal without an indication that the monitor has actually terminated.

`Last Start`

The date and time that the monitor was last instantiated.

Example

```
# ubmstat -vmn

                                          Num      Num      Num
                                          Times    Times    Times
Monitor   State [Ms/Mp] PSt    PID   Exit  Forked   Died     AWOL     Last Start
-------------------------------------------------------------------------------------------
g1n1m1    ACT   [ 1/2 ] RUN    29830 0     1        0        0        Mon Sep 30 14:02:10 2002
g1n1m2    SMART [ 1/2 ] RUN    29833 0     1        0        0        Mon Sep 30 14:02:10 2002
g1n2m1    UNREG [ 2/1 ] DEAD   0     0     0        0        0        N/A
g1n2m2    UNREG [ 2/1 ] DEAD   0     0     0        0        0        N/A
g2n1m1    ACT   [ 1/2 ] RUN    29834 0     1        0        0        Mon Sep 30 14:02:10 2002
g2n1m2    ACT   [ 1/2 ] DEAD   2642  0     123      0        0        Mon Sep 30 14:12:20 2002
g2n2m1    UNREG [ 2/1 ] DEAD   0     0     0        0        0        N/A
g2n2m2    UNREG [ 2/1 ] DEAD   0     0     0        0        0        N/A
```

OPTIONS (cont.)

For application information:

`PSt`

Process state of application.

`PID`

Process ID of application.

`Exit`

Last exit value of application.

`Num Times Forked`

The number of times that the application has been instantiated.

`Num Times Died`

The number of times that the application has unexpectedly terminated.

`Num Times AWOL`

The number of times that the application has been sent a termination signal without an indication that the application has actually terminated.

`Last Start`

The date and time that the application was last instantiated.

Example

```
# ubmstat -van
                                    Num       Num       Num
                                      Times     Times     Times
Appl                PSt    PID   Exit  Forked    Died      AWOL     Last Start
-------------------------------------------------------------------------------
-----------
ubapp11             RUN    276   0     3         0         0        Mon Sep 30
14:03:42 2002
ubapp12             RUN    279   0     3         0         0        Mon Sep 30
14:03:42 2002
ubapp21             DEAD   88    53248 1         0         0        Mon Sep 30
14:03:12 2002
ubapp22             DEAD   90    53248 1         0         0        Mon Sep 30
14:03:12 2002
ubapp23             DEAD   93    53248 1         0         0        Mon Sep 30
14:03:12 2002
```

-? (or *****)

This option prints the command usage.

FILES

None

SEE ALSO

None

17 ubManager Monitors

This chapter contains information about the use of monitors with ubManager. An alphabetically-ordered list of the monitors provided with ubManager and their functions is also provided.

In this chapter

Monitors Overview 203
loadMon 205
RPCmon 206
ubPinger 207

Monitors Overview

ubManager can use monitors to learn about the health of the system, essentially using upBeat as an event manager to determine whether or not user applications can run or remain running. Each CCPU-supplied monitor is designed to watch a particular element of the system and report status back to ubManager either directly through the monitor's exit status (for `RPCmon` and `loadMon`) or indirectly through upBeat (for `ubPinger`). In addition, you can write your own monitors to suit your needs; a monitor can be script or a binary executable; it can be an upBeat service or not.

If all of the monitors for a group indicate that the resources that they are monitoring are "healthy," ubManager will allow the group to remain active or to become active. However, if any one monitor of a group indicates that the resource that it is monitoring is not healthy, then ubManager will initiate a failover of the group if it is active, provided that its peer group's monitors indicate that all of the resources that they're monitoring are healthy. If a group is standby, ubManager will not allow the group to become active until all of its monitors indicate a healthy status for their resources.

ubManager supports two types of monitors: "smart" or "dumb".

- **Smart**
  A "smart" monitor is started by ubManager and interfaces directly with upBeat to update its state. ubManager tracks the monitor's active/standby state through ubManager's upBeat service callback. All smart monitors are considered permanent: if they should terminate, ubManager fails over its associated group (if it is active) and then restarts the monitor.

- **Dumb**
  A “dumb” monitor relies on ubManager to interface with upBeat on the monitor’s behalf. A dumb monitor can be permanent or periodic.
- *Permanent*
  These monitors start once and remain running. If a permanent monitor process exits while its corresponding group is active, ubManager will failover the group and restart the monitor. If a permanent monitor process exits while its group is standby, ubManager will restart the monitor and refuse any request to become active while the monitor is dead.
- *Periodic*
  These monitors run periodically, returning a status code each time they exit. If a periodic monitor exits with a non-zero value, ubManager considers the node “unhealthy” for the monitor’s associated group; and if a periodic monitor exits with a value of zero, ubManager considers the resource the monitor was monitoring as healthy.

Provided with the ubManager package are three monitors:

- `loadMon`
  Monitors a CPU’s load average.
- `RPCmon`
  Monitors RPC server programs (such as NFSd).
- `ubPinger`
  Monitors third-party IP addresses.

loadMon

loadMon enables the monitoring of the CPU's load average. loadMon is a dumb periodic monitor implemented as a script.

NAME

`load-mon`

USAGE

`/usr/lib/ubmgr/load-mon` ***`limit`***

DESCRIPTION

This utility is designed to monitor the current CPU's load average.

EXIT VALUES

Returns `0` if the usage is below the limit (success) and `1` if it is equal to or above (failure).

ARGUMENTS

`limit` is the CPU load average with which to compare the actual CPU load. It is expressed as a percentage in decimal form.

Example

```
load-mon 0.80
```

RPCmon

This utility monitors the status of RPC (remote procedure call) server programs, such as the network file system daemon (NFSd). RPCmon is a dumb permanent monitor implemented as a script.

NAME

rpcmon

USAGE

/usr/lib/ubmgr/rpcmon *type IP_address program version interval*

DESCRIPTION

This utility is designed to monitor an RPC process at a specified interval.

EXIT VALUES

Exits with a value of **1** if the RPC process does not respond; otherwise, the script does not normally exit.

ARGUMENTS

type is the protocol type and must be either **udp** or **tcp**.

IP_address is the IP address of the RPC server.

program is the name of the RPC program you want to monitor.

version is the version number of ***program***.

interval is the number of seconds between monitor calls to the RPC server.

Example

```
rpcmon udp localhost nfs 3 5
```

ubPinger

This utility monitors third-party IP addresses. ubPinger is a smart monitor, designed to monitor network connectivity and to cause a failover if an interface has failed.

ubPinger uses third-party addresses to determine whether a server has good connectivity to the network. These third-party addresses are machines out on the network that the server can use to determine if a path exists from the server to the network.

Careful choice of the third-party addresses is crucial. If an address is chosen and then goes off the network, it may trigger an improper failover. ubPinger allows you to specify a number of addresses to check to reduce the chance of improper failover. If any one of the addresses can be reached, the interface is presumed functional.

NAME

ubpinger

USAGE

/usr/lib/ubmgr/ubpinger [-t *ping_time*] *service_name host1* [*host2*]

DESCRIPTION

This utility is designed to monitor network connectivity and cause a failover if an interface has failed.

OPTIONS

ping_time is the time between pings, in milliseconds. The default value is 500 ms.

ARGUMENTS

service_name is the name of the service that `ubPinger` becomes to report success or failure. This is defined under a `<SERVICE>` tag of the upSuite configuration file, `upsuite.conf`.

host1 is the IP address or host (specified with the hostname) you want to monitor.

host2 is a second IP address or host (specified with the hostname) you want to optionally monitor.

Example

```
ubpinger server1-net1 -t 400 192.168.1.200 192.168.1.201
```

18 ubManager Semicolon-Delimited Configuration File

ubManager supports two different formats for its configuration file: an XML format, and the previously supported semicolon-delimited file format. The XML format is preferred. This chapter describes the semicolon-delimited file format, which is still supported for the purposes of backwards compatibility only.

ubManager includes a sample semicolon-delimited configuration file, `/etc/upsuite/examples/ubmgr.sc`. A semicolon-delimited configuration file consists of the following:

- A version string (required)
- Zero or more group definitions
- Zero or more application definitions
- Zero or more monitor definitions

Each of these parameters is detailed below.

Comments

Lines beginning with an octothorpe (#) are considered comments and are ignored by the program. The exception to this rule is that the first characters of the configuration file must be as follows: **`#:3`**.

Version

The first line of `ubmgr.conf` must contain a version string so ubManager can tell the file format. The first characters of the file must be an octothorpe (#), a colon (:), and the version number of the ubManager semicolon-delimited configuration file format; only version 3 is supported.

Your first line must look similar to the following:

```
#:3 Configuration File Format 3 (Don't change this line!)
```

Group Definition

The **g** parameter defines a group, which must have a corresponding <SERVICE> element in the upSuite configuration file, `/etc/upsuite/upsuite.conf`. If the group is to follow a lead service, then a lead service must also be specified, which must also have a corresponding <SERVICE> element in the upSuite configuration file.

USAGE

g;*group_name*;[*lead_service*]

DESCRIPTION

This entry defines a group.

The ***group_name*** and ***lead_service*** (which is optional) must be defined as services in the `upsuite.conf` configuration file.

If ***lead_service*** is defined, ubManager will control ***group_name***'s state so that it tracks that of ***lead_service***. If ***lead_service*** is not defined, the group will become active on the node on which ubManager runs first.

Example

```
g;group1;updisk:/ipfs
```

Monitor Definition

The **m** parameter defines a monitor associated with a group. Each monitor must have a corresponding <SERVICE> element in the upSuite configuration file for the node it is to run on.

USAGE

```
m;group_name;node_id;monitor_service;cmdline;[options]
```

DESCRIPTION

This entry defines a monitor process.

A monitor is associated with a particular group, as specified by ***group_name***, and with a particular node, as specified by ***node_id***.

Monitors require a unique service definition in `upsuite.conf` *for each node*. This is so that ubManager can concurrently keep track of the groups' monitors on both nodes.

The unique service name for the node is supplied in the ***node_id*** and ***monitor_service*** parameters.

A ***cmdline*** is also required so ubManager can start and stop the monitor process at the appropriate times.

The available ***options*** include:

`interval` ***n***

This would cause a restart of a dumb monitor every ***n*** seconds. This defines the monitor as a dump periodic monitor.

`smart`

This would cause ubManager to monitor the monitor's state through upBeat.

If there is no interval option and no smart option, then the monitor is considered to be a dumb permanent monitor.

Note: The command line arguments should not be separated by semicolons, only white spaces.

Example

```
m;group1;1;server1m2;monitors/rpcmon udp localhost nfs 3 5
```

Application Definition

The **a** parameter defines an application associated with a group.

USAGE

a;*group_name*;*appl_name*;*cmdline*

DESCRIPTION

This entry defines a user application process that should always be running when ***group_name*** is active.

A unique name for this process must be supplied in ***appl_name***.

Finally, a ***cmdline*** must be given so that ubManager can start and stop the application process at the appropriate times. If the user application process exits, ubManager will initiate group recovery (failover).

Note: The command line arguments should not be separated by semicolons, only white spaces.

Example

```
a;group1;ubapp;/usr/lib/ubmgr/ubapp
```

Sample Semicolon-Delimited Configuration File

Following is the a sample configuration file in semicolon-delimited format (this is in the `/etc/upsuite/examples/ubmgr.sc` file).

```
#:3 Configuration File Format 3 (Don't change this line!)

# Group1
g;group1
m;group1;1;g1n1m1;/usr/lib/ubmgr/load-mon 0.80;interval 5
m;group1;2;g1n2m1;/usr/lib/ubmgr/load-mon 0.80;interval 5
a;group1;g1a1;/usr/lib/ubmgr/ubapp1 arg1 arg2
a;group1;g1a2;/usr/lib/ubmgr/ubapp2 -gar arg
a;group1;g1a3;/usr/lib/ubmgr/ubapp3

# Group2
g;group2;updisk:/mydata
m;group2;1;g2n1m1;/usr/lib/ubmgr/rpcmon udp machine1 rpcapp 3 5
m;group2;2;g2n2m1;/usr/lib/ubmgr/rpcmon udp machine2 rpcapp 3 5
m;group2;1;g2n1m2;/usr/lib/ubmgr/ubpinger -t 1000 g2n1m2 machine1;smart
m;group2;2;g2n2m2;/usr/lib/ubmgr/ubpinger -t 1000 g2n2m2 machine2;smart
a;group2;g2a1;/usr/lib/ubmgr/ubapp4
```

```
a;group2;g2a2;/usr/lib/ubmgr/ubapp5

# Group3
g;group3;updisk:/ipfs
```

The sample `upsuite.conf` file, `/etc/upsuite/examples/upsuite.4ubmgr.xml`, shown in the XML configuration file section also applies to `/etc/upsuite/examples/ubmgr.sc`.

Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	Courier font indicates the names of commands, files and directories, and on-screen computer output	Edit your `.login` file. At the `ok` prompt….
`AaBbCc123`	Bold Courier font indicates a command you type, contrasted with on-screen computer output. Also used for command arguments.	To turn the unit on, type **`on`** at the `ccpu>` prompt. `ccpu>:`**`on`**
`AaBbCc123`	Bold italics indicate a command-line placeholder or token to be replaced with a real name or value	To delete a file, type **`rm`** ***`filename`***.
[**`AaBbCc123`**]	Square brackets indicate an optional argument (do not type brackets)	[**`help`**] **`dir`** [***`filename`***]
{***`a`*** **`|`** ***`b`***}	Curly braces indicate a choice of required argument (do not type brackets). The vertical line | separates the choices. You must choose one and only one of the items.	**`grade`** {**`a|b|c|d|f`**}
AaBbCc123	Italics indicate book titles, new words or terms, or words to be emphasized	• This manual is used in conjunction with the *SPARCengine CP1500 User's Manual.*
`Ctrl`	Keystroke press	Send a break using **`Ctrl-]`**.
	Caution	Failure to heed the instructions that follow the Caution symbol may result in damage to the equipment.

Glossary

Active (service)	The service that can read and write to the file system.
Client	An application that uses upBeat in order to monitor the network.
Failover	The migration of applications or services from one machine to another in the event of failure.
HA NFS	Abbreviation for a high availability implementation of NFS (see the NFS definition below). A pair of standard NFS servers, plus upSuite HA software, are used to build a high availability NFS server.
`ipfs`	The kernel module of upDisk.
Local	The current system on which upBeat resides.
Mean Time to Repair (MTTR)	The average time it takes to complete repairs on equipment or services.
NFS	A computer industry-standard network file system protocol developed for distributing files within a heterogeneous network.
Node	Any processor running upSuite HA.
Peer	The partner of the current local node.
SCSIbeat	SCSIbeat is a component of upBeat that monitors disk failures.
Split Brain	All network connectivity between two nodes has been lost, often resulting in multiple active services.
Standby (service)	The service that is ready to take over if the active service fails. This service can read the file system, but no changes can be made. If the active fails, the applications can be failed over to the standby using the replicated data.
UFS	Abbreviation for UNIX file system.
VFS	Abbreviation for virtual file system.

Technical Support

Before contacting the Technical Support team at Continuous Computing, be sure you have read this manual carefully.

If you continue to experience problems, please contact the Technical Support team at Continuous Computing by any of the methods listed below.

Please be sure to include the serial numbers for each affected module, system and/or part. In addition, we will need to know what version of Solaris you are running, as well as the patch level, and any other significant software packages that are installed.

To contact the Technical Support team at Continuous Computing, do one of the following:

- Email us at support@ccpu.com
- Visit our support web site at http://support.ccpu.com

  (This site features our automatic technical support system. Create a new user profile. Then submit a new ticket at the “Welcome to SupportWizard” page. This process ensures that our team delivers a timely solution to any technical problem you have.)
- Call us at (858) 882-8911, 9:00 a.m. – 5:00 p.m. (PST)

Note: If you have a Gold or Platinum service contract, follow the contact instructions provided with your contract.

Index

Directories & Files

/etc/dfs/dfstab 121
/etc/ipfstab 117, 161
/etc/upsuite 117, 146
/etc/upsuite.conf 146
/var/log/nidb 89
/var/log/upsuite 15

Configuration Tags

< LINK> 156
<HANFS> 157
<HEARTBEAT> 152
<NETWORK> 149
<NODE> 149
<NODE_REF> 155
<SCSIBEAT> 157
<SERVICE> 154
<UPBEAT> 148
<UpSuiteConfig> 148

A

Active (definition) 217
API 23, 33
    function calls 35
    overview 23
    quick reference 35
Application write operations 85
Applications
    local 10
    multi-threaded 24

B

benefits 177, 179

C

Changing mount options 123
Client (definition) 217
command line 211
Commands 17
    ubManager CLI 187
    udactive 128
    udrepair 131
    udstat 132
    upbeat (admin) 17
    updisk (admin command) 137
    updisk (startup script) 139
    ups 21
Configuration file 145
    changes 102
    editing 145, 147
    example 159
    ubmgr.conf
        application definition 212
        comments 209
        group definition 210
        monitor 210
        version 209
Configurations
    sample A 161
    sample B 166
    sample C 170
    sample partition table 175
Configuring services 154
Console messages 89
    active/standby errors 93
    bringing systems online 89
    comments 92
    link status 92
    miscellaneous transmission errors 94
    mismatches 90

operational messages 91
operator intervention 90
problems with startup exchange 89
role status 91
starting up and mounting ipfs 91
state status 92
warning messages, underlying file system 90
Ctrl-C 58

D

Datasets 104
in ipfstab 118
sharing in ipfstab 112
Debug mode 15, 18, 19, 138
Definitions of Terms 217
Disk space 95

E

Editing the configuration file 102, 111, 145, 147
<SERVICE> tag 112

F

Failover 7, 102, 154
Failover (definition) 217
features 81, 177, 179
File locking 102
File modification times 96
File systems 95
access denied 94
underlying 118, 119
FSYNC 122
Function calls 35
guidelines 23
ubAckSvc( ) 36
ubAsync( ) 37
ubFini( ) 40
ubGetState( ) 41
ubInit( ) 43
ubNode( ) 44
ubNodeName( ) 45
ubRegSvc( ) 46
ubServiceIP( ) 47
ubSetupPollfd( ) 48
ubSvc( ) 49
ubSvcIPPair( ) 50
ubSvcName( ) 52
ubSvcPeer( ) 53
ubSvcPort( ) 54

H

HA NFS 107
subtag 157
HA NFS (definition) 217

I

Interface 111
ioctl function call 141
IP addresses 111, 146, 150, 154, 158
ipfs 84, 217
mount options 118
mount point 118
mounting 91
operations 85
starting up 91
unmounting 95
ipfstab 117

K

kstat 89

L

LANs 15, 171
libupbeat 23

Local (definition) 217
Local applications 10
Local file system
    device 119
    mount options 120
    mount point 118
    type 118
Locking
    files 102

M

MAC addresses 15
Managed failover 102
MANs 153, 171
Metadevice 166
Monitoring SCSI disks 157
Monitoring upBeat 15
Monitoring upDisk 89
monitors
    dumb 204
    periodic 204
    permanent 204
    smart 203
Mount options 121
    changing 123
    maximum operations 123
    memory allocation 124
Multi-threaded applications 24

N

Nodes 217

O

O_SYNC 122
Online Disk Suite 166, 175
Operation overview 10
Operations 82
    overview 82
    write 85
overview
    ubManager 180

P

Packages
    adding 13
Partition table 175
Partitions 149, 151
    SCSI disks 149
Partitions.See also SCSIbeat
Peer (definition) 217

R

Registering services (API) 23
Repair 85
    failure 96
Replay 85
RESEND_MSEC attribute 153
Restarting 102, 147
ROUTE attribute 153
Routing 14, 152, 153, 154, 171

S

sample (sample API program) 64
Sample API programs
    multi-threaded 69
    status checker 64
        source code 65
    wrapper 57
        source code 58
Sample configurations 161
SCSI disks 149, 151
SCSIbeat 11, 16, 151, 217
    sample subtag 154
Service tag of upsuite.conf 112

Services
configuring 154
registering 23
Share options 121
Sharing datasets in ipfstab 112
Sockets 152, 153
Split brain 13, 14, 97, 101, 153
definition 217
recovering from 97
Standby 217
Starting upBeat 19
Starting upDisk 139
status (API program)
source code 65
Stopping upBeat 19
Stopping upDisk 139
syslog 15, 89, 138
System administration 13, 87

T

TCP 111, 114
technical support 219
Timeout_msec attribute 153
Troubleshooting
conflicting file modification times 96
file system access denied 94
file system out of disk space 95
HA NFS failover 99
ipfs unmount unsuccessful 95
monitoring upDisk 89
split brain conditions 97
Typographic conventions 215

U

ubAckSvc( ) 36
ubAsync( ) 37
ubFini( ) 40
ubGetState( ) 41
ubInit( ) 43
ubManager 10
ubNode( ) 44
ubNodeName( ) 45
ubRegSvc( ) 46
ubServiceIP( ) 47
ubSetupPollfd( ) 48
ubSvc( ) 49
ubSvcIPPair( ) 50
ubSvcName( ) 52
ubSvcPeer( ) 53
ubSvcPort( ) 54
udactive 128
UDP 111, 114
udrepair 131
udstat 132
Underlying file systems 88, 118, 119
Unmounting ipfs 95
upBeat
benefits 6
features 6
maintenance 15
major functions 6
monitoring 15
network operation 3, 7
starting 19
stopping 19
upbeat (admin command) 17
upbeat (startup/shutdown script) 19
upDisk
benefits 81, 82
features 81
operations 82
starting 139
stopping 139
updisk (admin command) 137
updisk (startup script) 139
uplicense 20
upsuite.conf 145
editing 147
example 159

V

Virtual LANs (VLANs) 15

W

WANs 153, 171
wrapper (sample API program) 57
    source code 58
Write operations 85

X

XML 145, 146
    attributes 146